# Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Spring & Summer 2024
Lecture 2
2024-3-4

# Outline:
# Balanced Search Trees (II)

- Review of amortized analysis
- Red-black trees
- B+ trees
- Take-home messages

# Outline:
# Balanced Search Trees (II)

- Review of amortized analysis
- Red-black trees
- B+ trees
- Take-home messages

# Amortized Analysis

# Amortized Analysis

Target : Any $M$ consecutive operations take at most $O(M \log N)$ time.

# Amortized Analysis

Target : Any $M$ consecutive operations take at most $O(M \log N)$ time.

-- *Amortized* time bound

# Amortized Analysis

Target : Any $M$ consecutive operations take at most $O(M \log N)$ time.

-- *Amortized* time bound

worst-case bound  amortized bound  average-case bound

# Amortized Analysis

Target : Any $M$ consecutive operations take at most $O(M \log N)$ time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound average-case bound

# Amortized Analysis

Target : Any $M$ consecutive operations take at most $O(M \log N)$ time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound ≥ average-case bound

# Amortized Analysis

Target : Any $M$ consecutive operations take at most $O(M \log N)$ time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound ≥ average-case bound

Probability
is *not* involved

# Amortized Analysis

Target : Any $M$ consecutive operations take at most $O(M \log N)$ time.

-- *Amortized* time bound

worst-case bound ≥ amortized bound ≥ average-case bound

Probability
is *not* involved

☞ Aggregate analysis

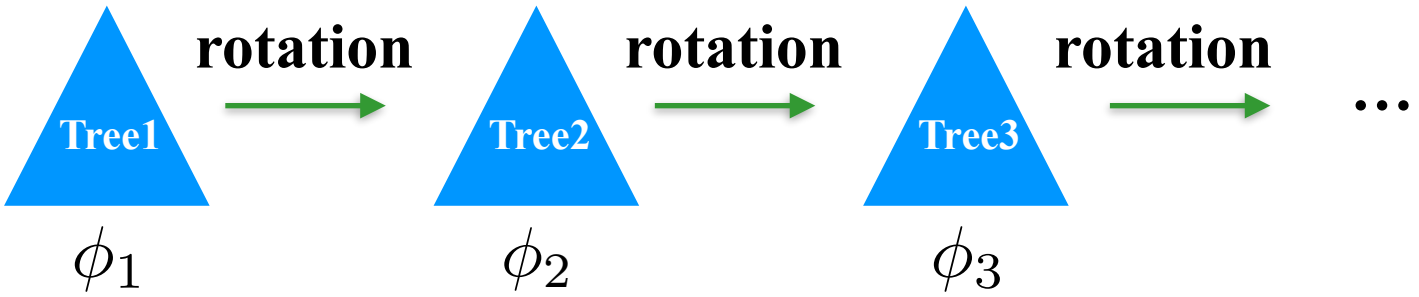
☞ Accounting method

☞ Potential method

# Analysis of Splay Trees

- What we want to bound?
  - The amortized cost of a sequence of operations, e.g. search, delete, insert, split…
  - Each operation involves slaying: a subsequence of rotations.

- The potential function is built on a state of tree. Let's consider the amortized cost of sequence of rotations first.



〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

$D_i =$

$\Phi(D_i) =$

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

$D_i$ = the root of the resulting tree

$\Phi(D_i) =$

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

$D_i$ = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over $n$ steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

$D_i$ = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over $n$ steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

where $S(i)$ is the number of descendants of $i$ ($i$ included).

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

$D_i$ = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over $n$ steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

where $S(i)$ is the number of descendants of $i$ ($i$ included).

*Rank of the subtree*
*≈ Height of the tree*

〖Example〗 Splay Trees: $T_{amortized} = O(\log N)$

$D_i$ = the root of the resulting tree

$\Phi(D_i)$ = must increase by at most $O(\log N)$ over $n$ steps, AND will also cancel out the number of rotations (*zig*:1; *zig-zag*:2; *zig-zig*:2).

$$\Phi(T) = \sum_{i \in T} \log S(i)$$

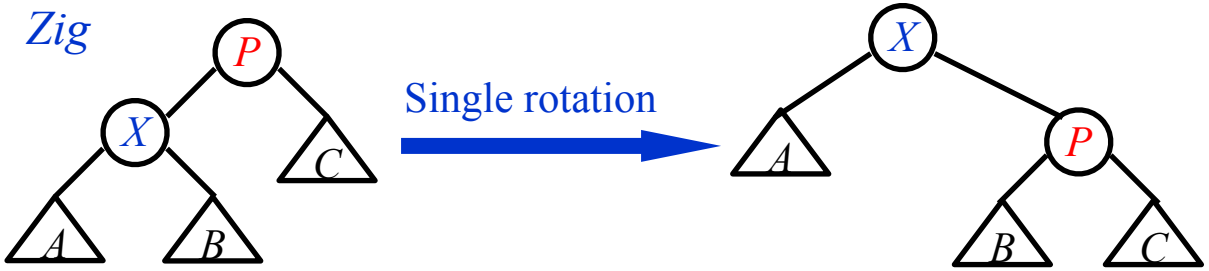
where $S(i)$ is the number of descendants of $i$ ($i$ included).

*Rank of the subtree*
*≈ Height of the tree*

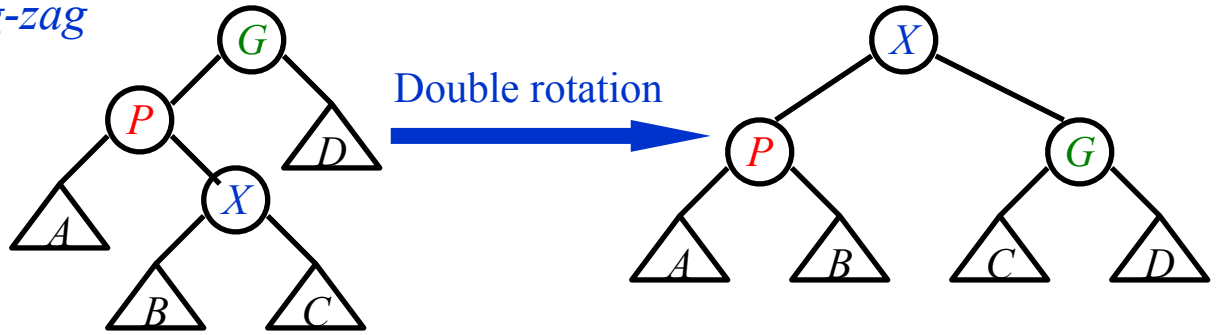
Why not simply use the heights of the trees?

$$\Phi(T) = \sum_{i \in T} Rank(i)$$

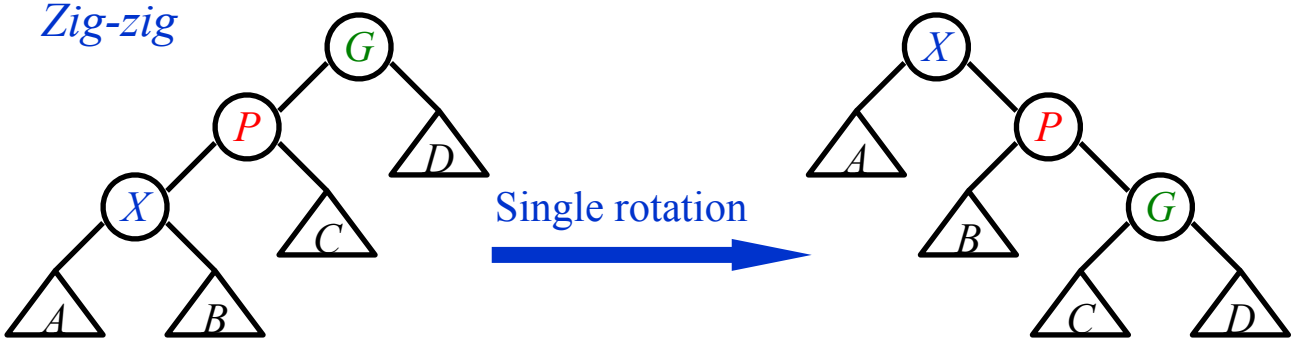
*Zig*




*Zig-zag*

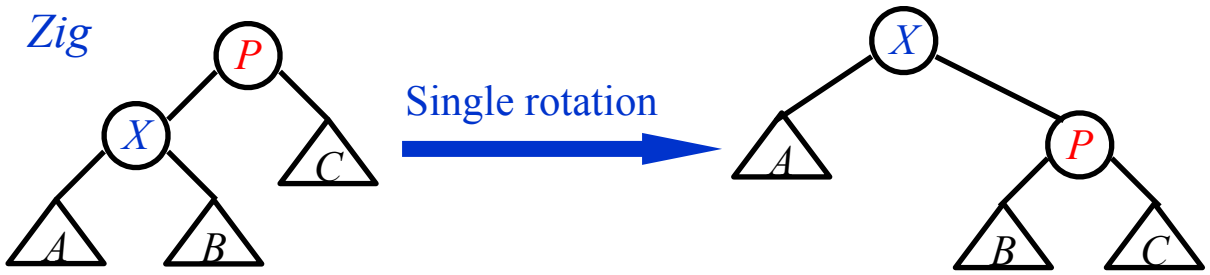



*Zig-zig*



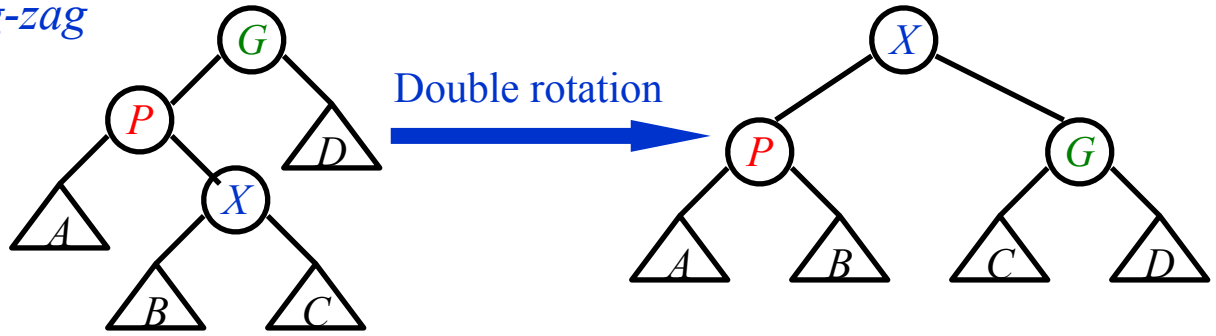
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*

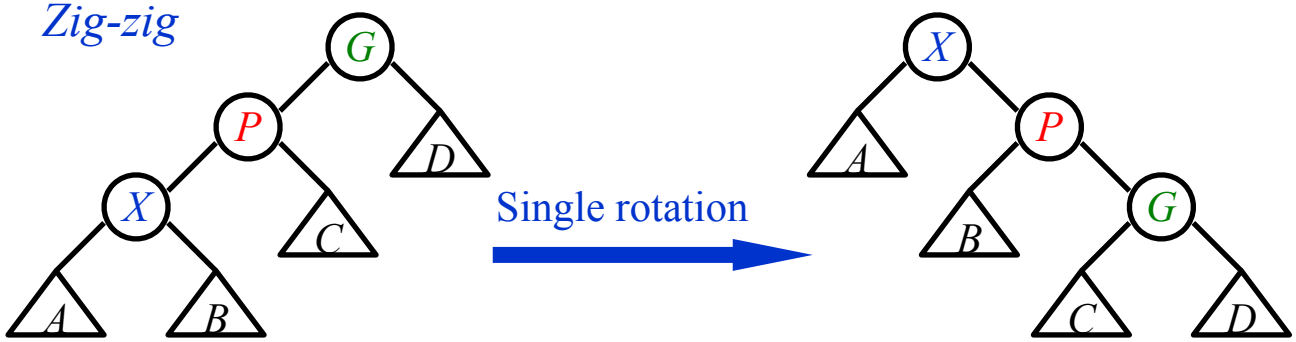



$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*

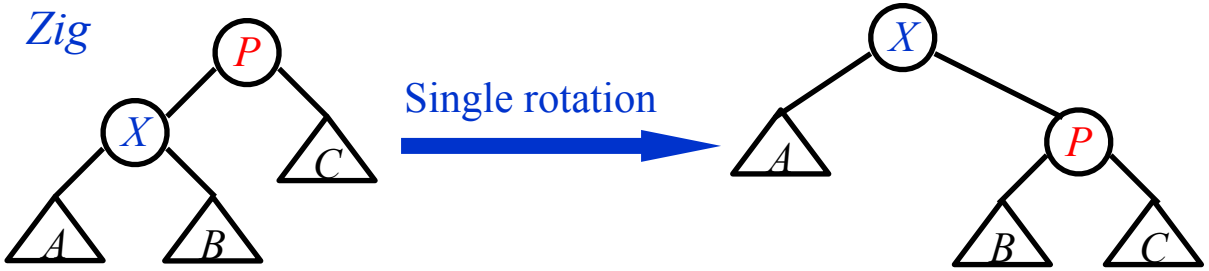



*Zig-zig*



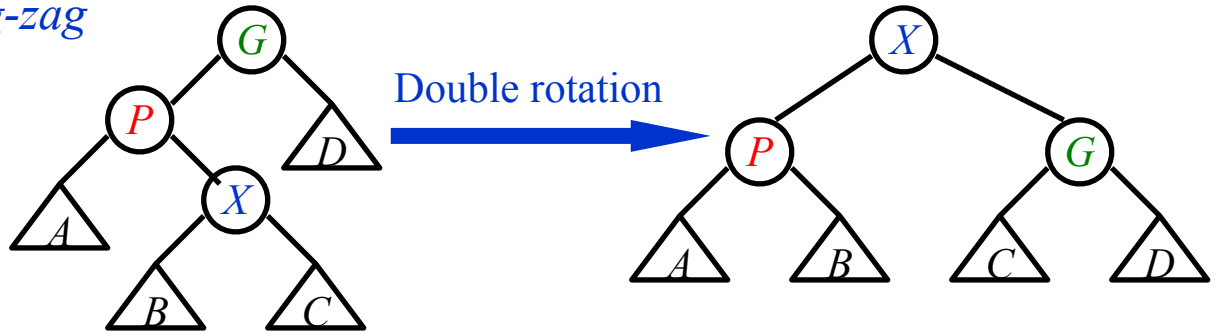
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*

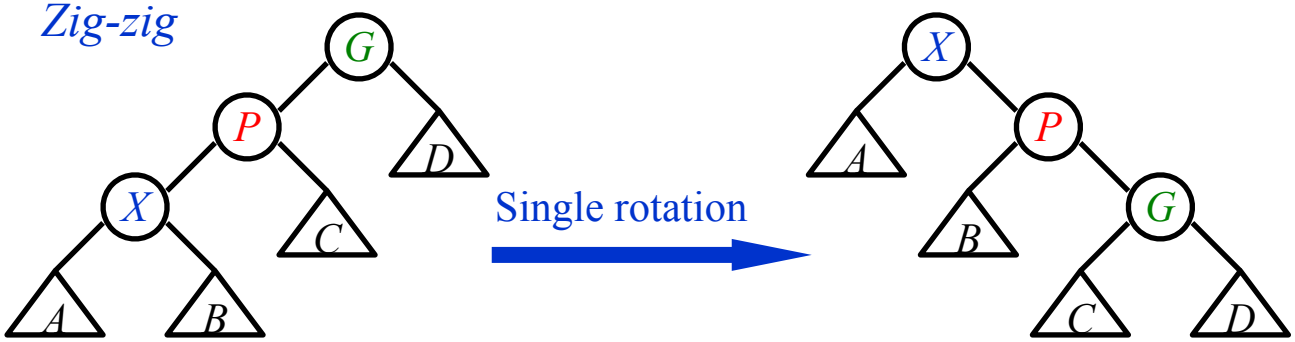



$$\hat{c}_i = 1 + R_2(X) - R_1(X)$$
$$+ R_2(P) - R_1(P)$$
$$\leq 1 + R_2(X) - R_1(X)$$

*Zig-zag*

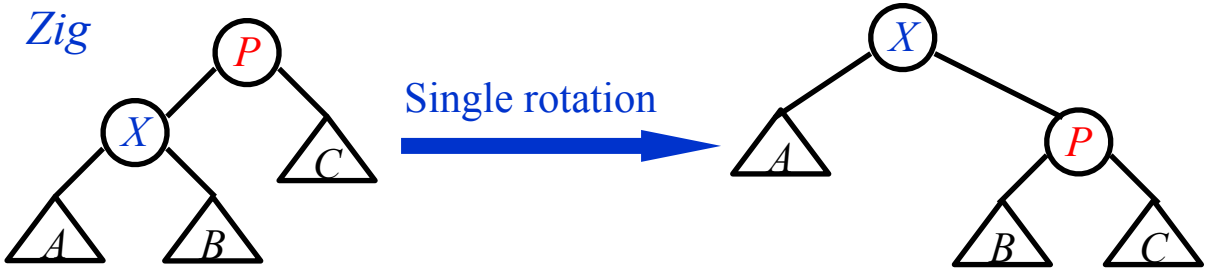



*Zig-zig*



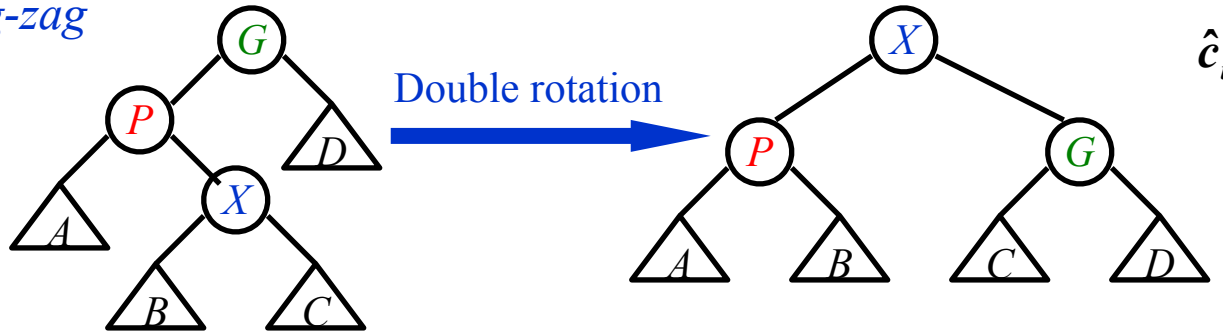
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*



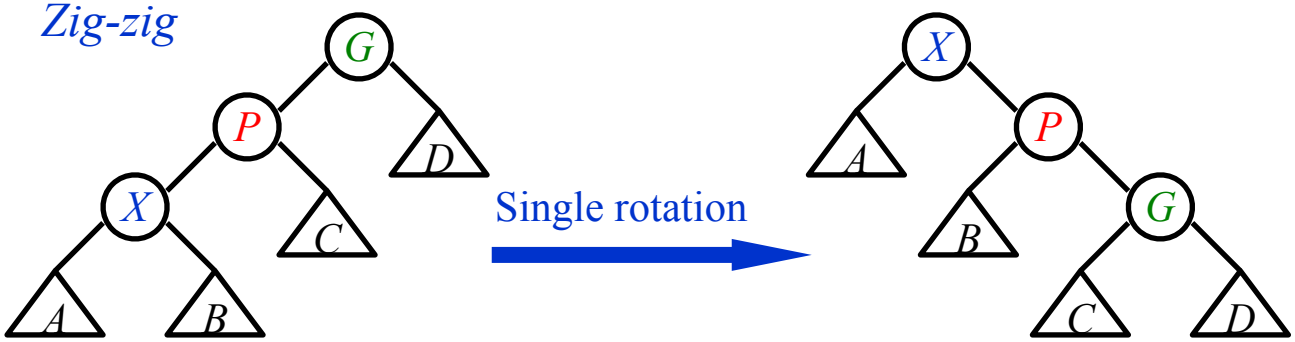

$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*



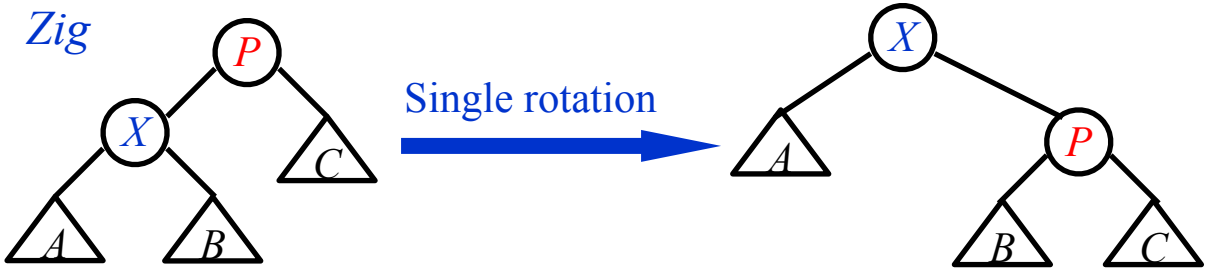

$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

*Zig-zig*



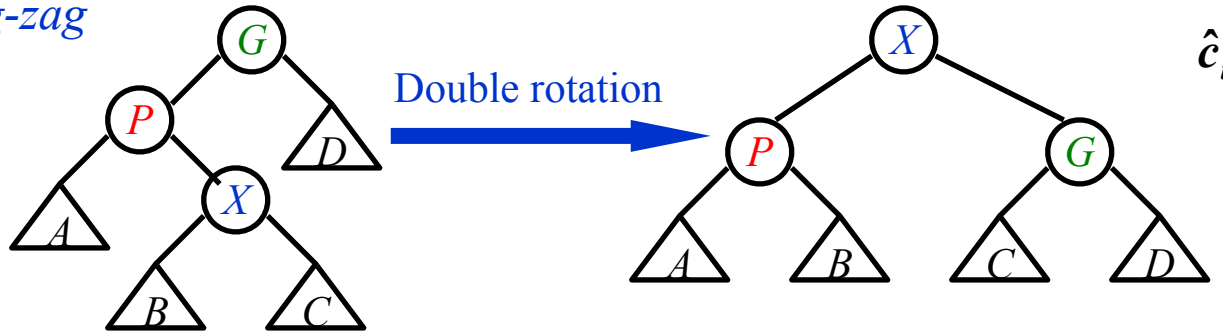
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*



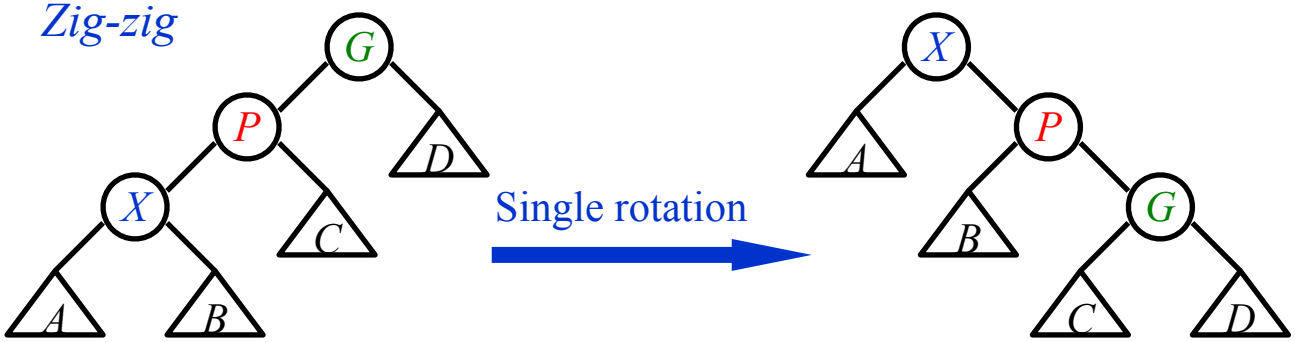

$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*



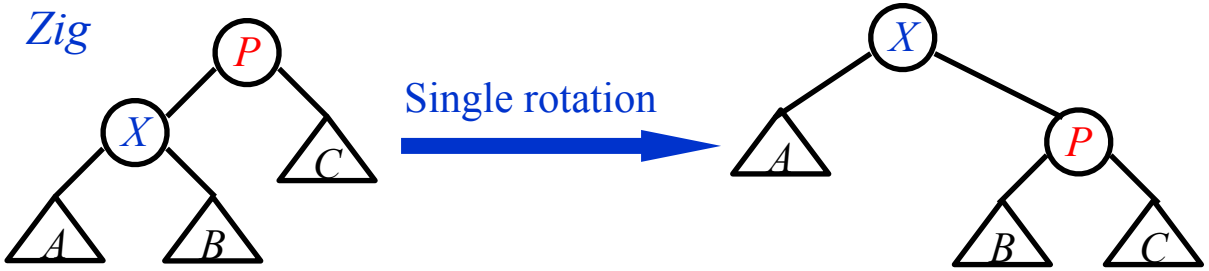

$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

*Zig-zig*



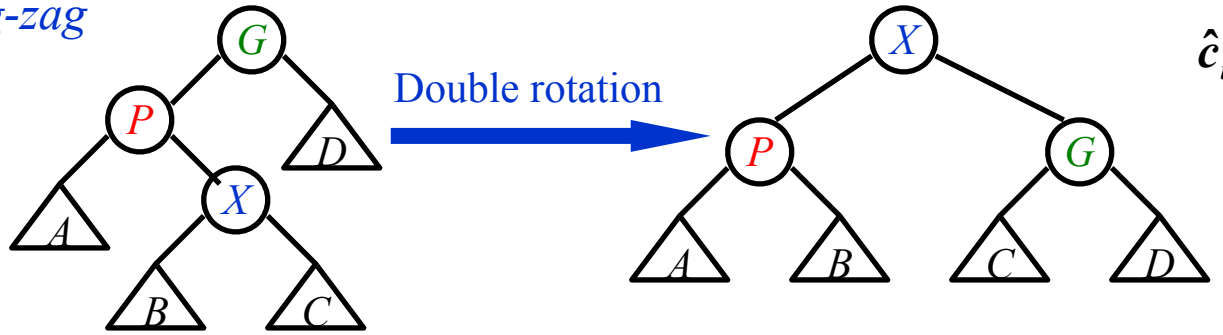
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*



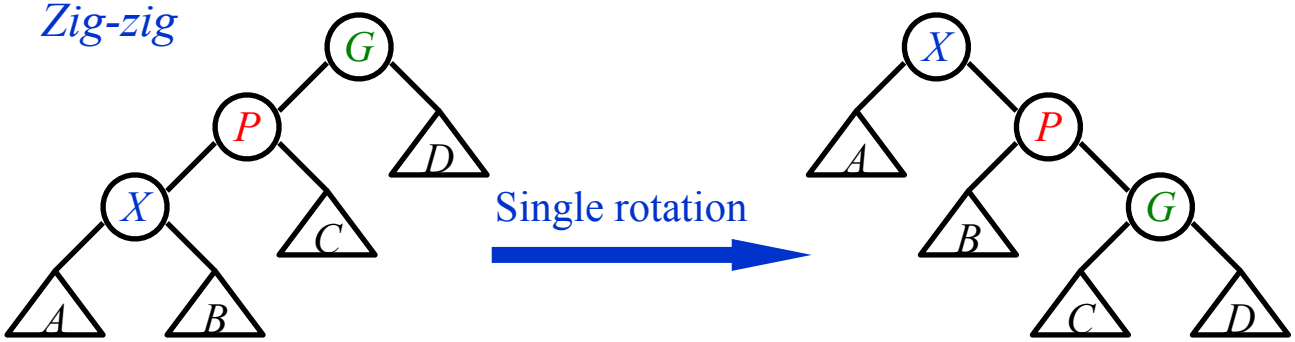

$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*



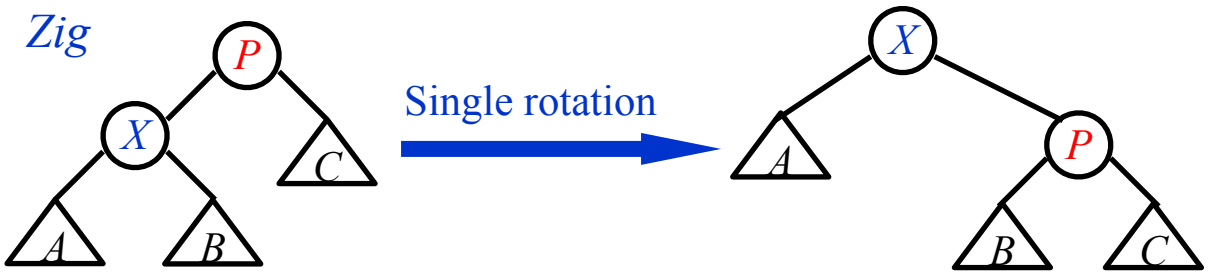

$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

*Zig-zig*



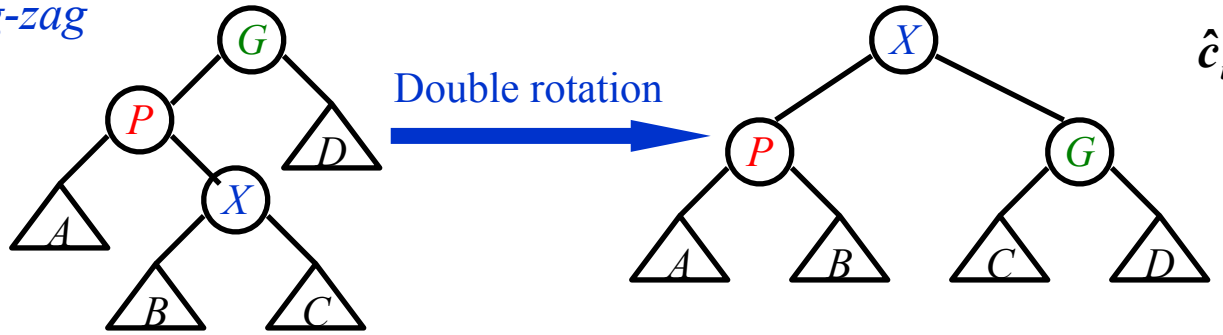
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*




$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*

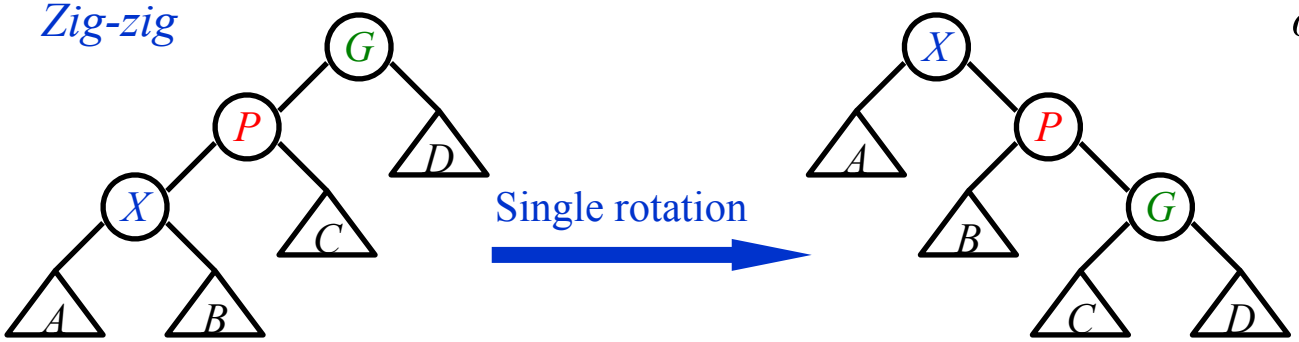



$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

Lemma 11.4 on [Weiss] p.448

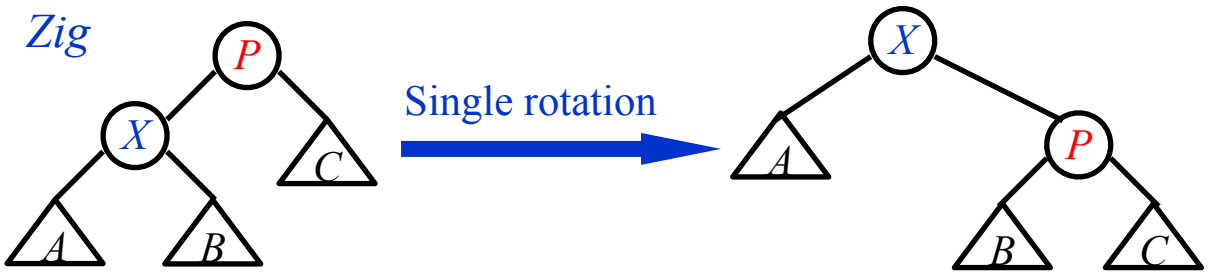
$$a + b \le c \Rightarrow ab \le c/4$$

*Zig-zig*



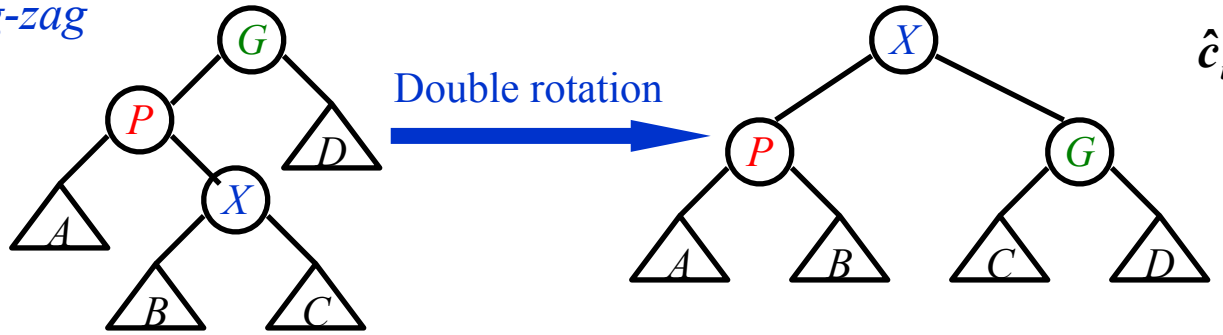
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*




$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*

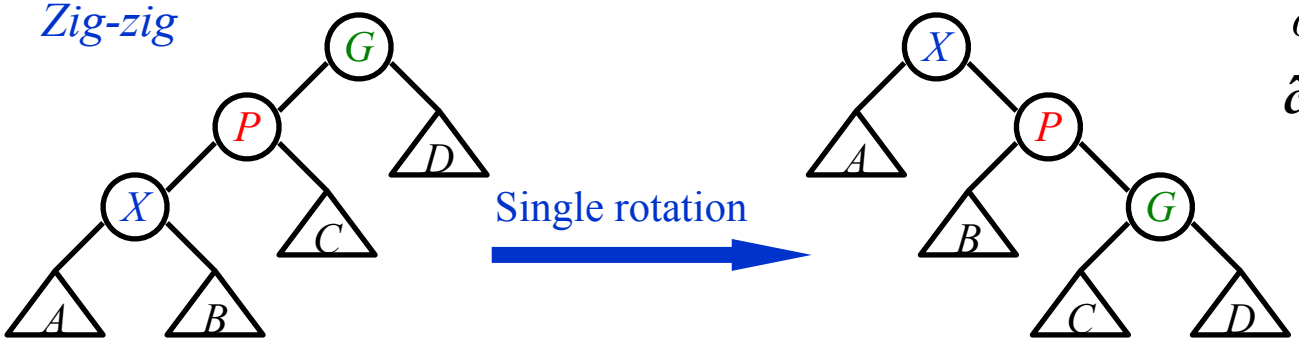



$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

Lemma 11.4 on [Weiss] p.448

$$a + b \le c \Rightarrow ab \le c/4$$

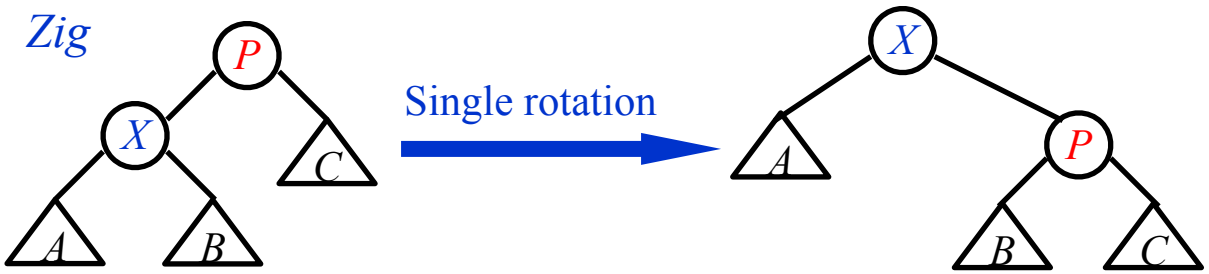
*Zig-zig*




$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 3\left(R_2(X) - R_1(X)\right)$$

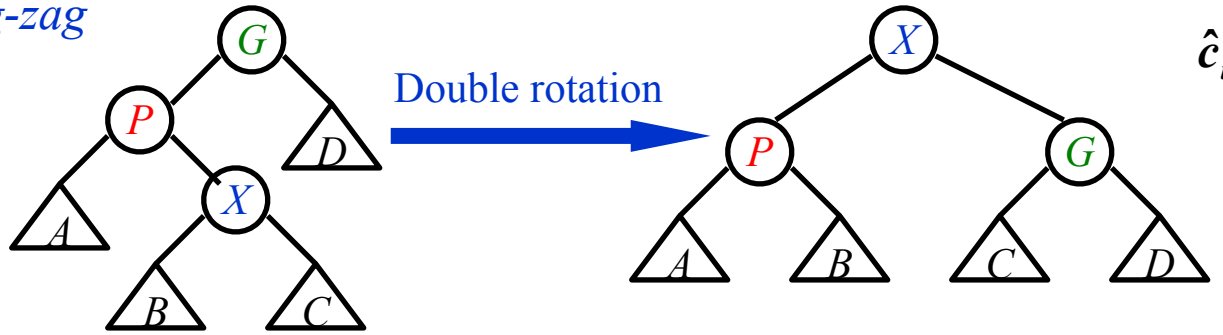
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*




$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*

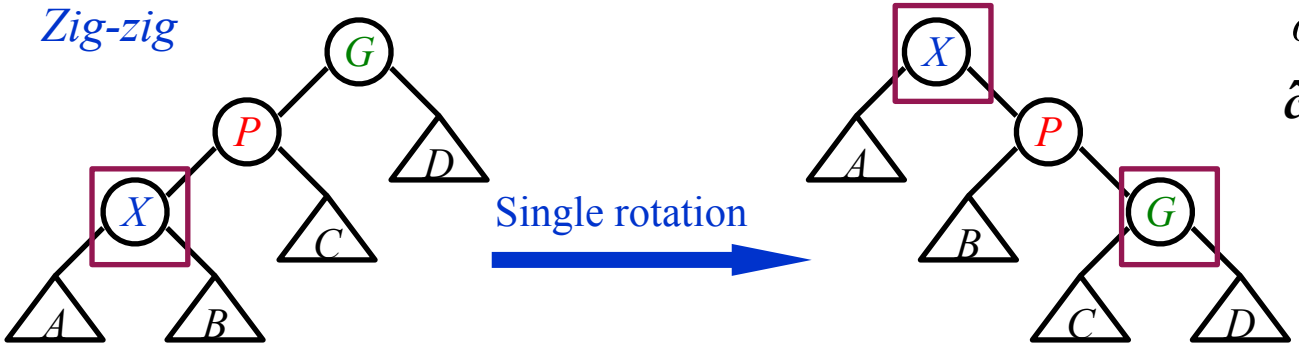



$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

Lemma 11.4 on [Weiss] p.448

$$a + b \le c \Rightarrow ab \le c/4$$

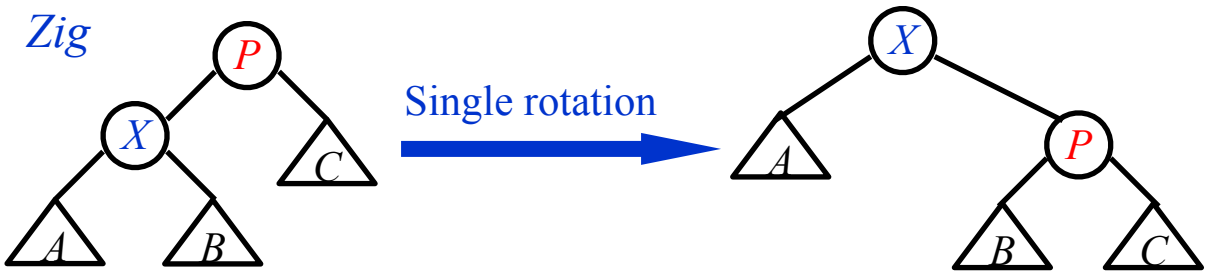
*Zig-zig*




$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 3\left(R_2(X) - R_1(X)\right)$$

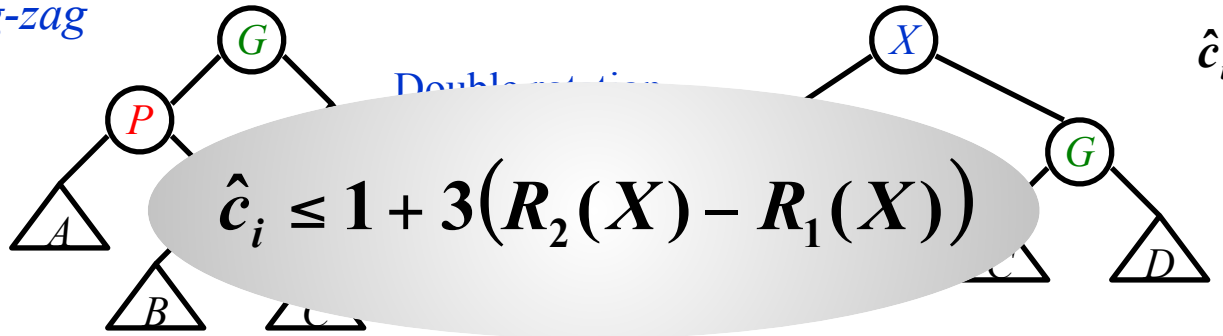
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*




$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*



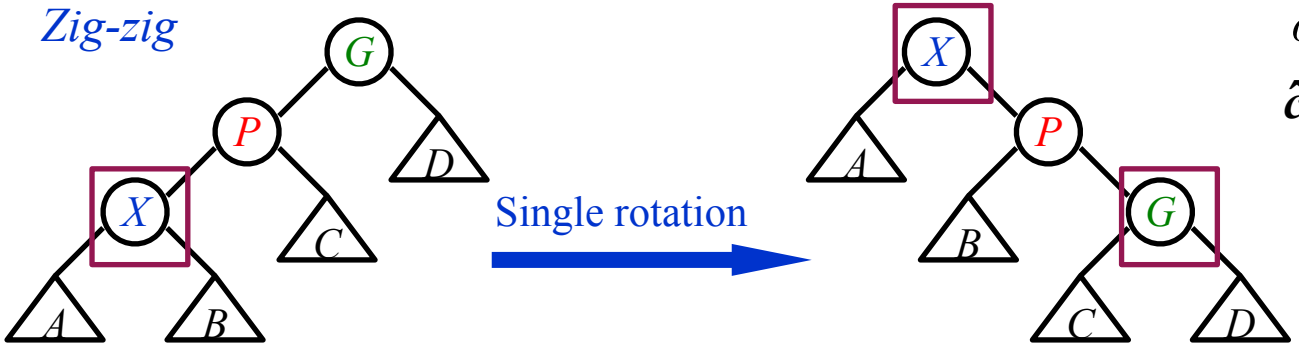

$$\hat{c}_i \le 1 + 3\left(R_2(X) - R_1(X)\right)$$

$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

Lemma 11.4 on [Weiss] p.448

$$a + b \le c \Rightarrow ab \le c/4$$

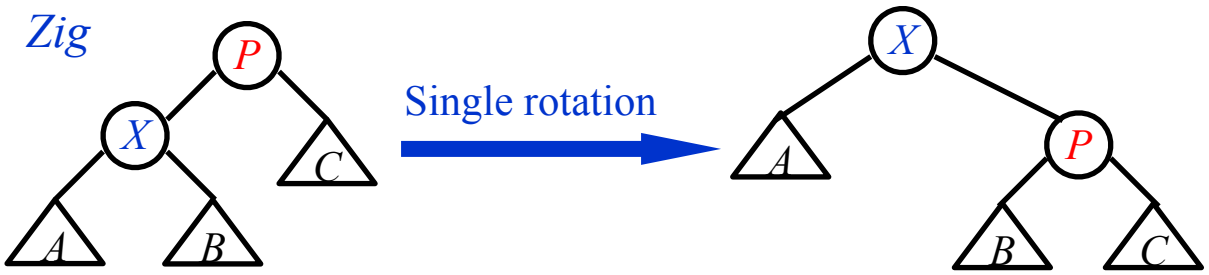
*Zig-zig*




$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 3\left(R_2(X) - R_1(X)\right)$$

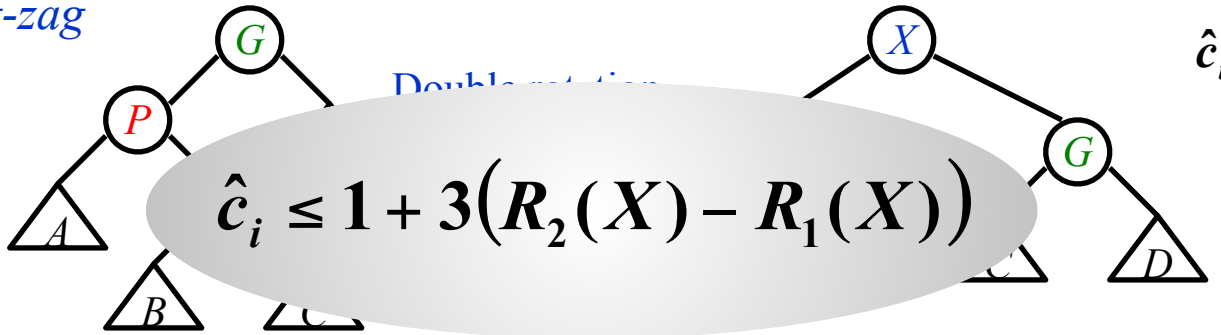
$$\Phi(T) = \sum_{i \in T} Rank(i)$$

*Zig*




$$\hat{c}_i = 1 + R_2(X) - R_1(X) + R_2(P) - R_1(P) \le 1 + R_2(X) - R_1(X)$$

*Zig-zag*



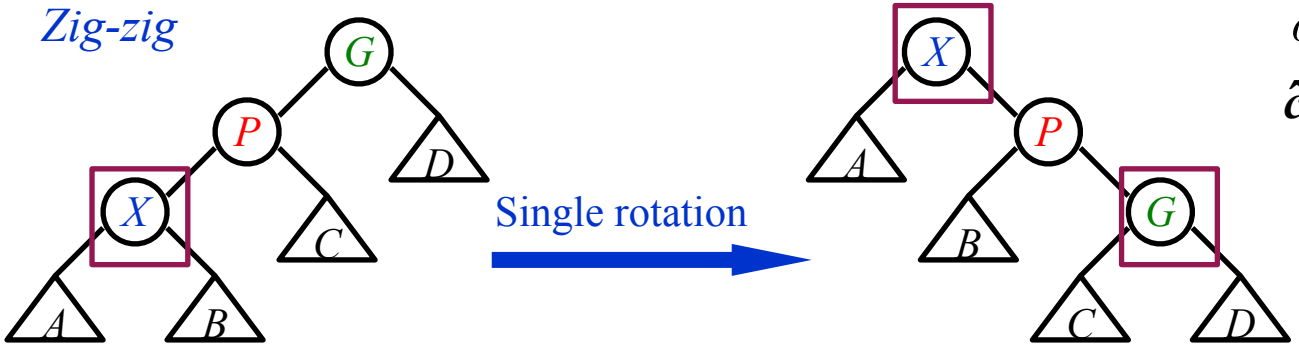

$$\hat{c}_i \le 1 + 3\left(R_2(X) - R_1(X)\right)$$

$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 2\left(R_2(X) - R_1(X)\right)$$

Lemma 11.4 on [Weiss] p.448

$$a + b \le c \Rightarrow ab \le c/4$$

*Zig-zig*




$$\hat{c}_i = 2 + R_2(X) - R_1(X) + R_2(P) - R_1(P) + R_2(G) - R_1(G) \le 3\left(R_2(X) - R_1(X)\right)$$

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root $T$ at node $X$ is at most $3(R(T) - R(X)) + 1$.

# Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root $T$ at node $X$ is at most $3( R( T ) - R ( X ) ) + 1$.

# Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root $T$ at node $X$ is at most $3(R(T) - R(X)) + 1$.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \left(\sum_{i=1}^{n} c_i\right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

# Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root $T$ at node $X$ is at most $3(R(T) - R(X)) + 1$.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \left(\sum_{i=1}^{n} c_i\right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Should assume to start from an empty tree

# Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root $T$ at node $X$ is at most $3(R(T) - R(X)) + 1$.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \left(\sum_{i=1}^{n} c_i\right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Should assume to start from an empty tree

We should also consider the influences of other steps other than rotations on the potential functions.
Fortunately, their influences are minor.

# Amortized Cost of Splay Trees

【Lemma】 The total cost of $\sum \hat{c}_i$ to splay a tree by a series of rotations with root $T$ at node $X$ is at most $3(R(T) - R(X)) + 1$.

bounded by log(N)

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \left(\sum_{i=1}^{n} c_i\right) + \underbrace{\Phi(D_n) - \Phi(D_0)}_{\geq 0}$$

Should assume to start from an empty tree

We should also consider the influences of other steps other than rotations on the potential functions.
Fortunately, their influences are minor.

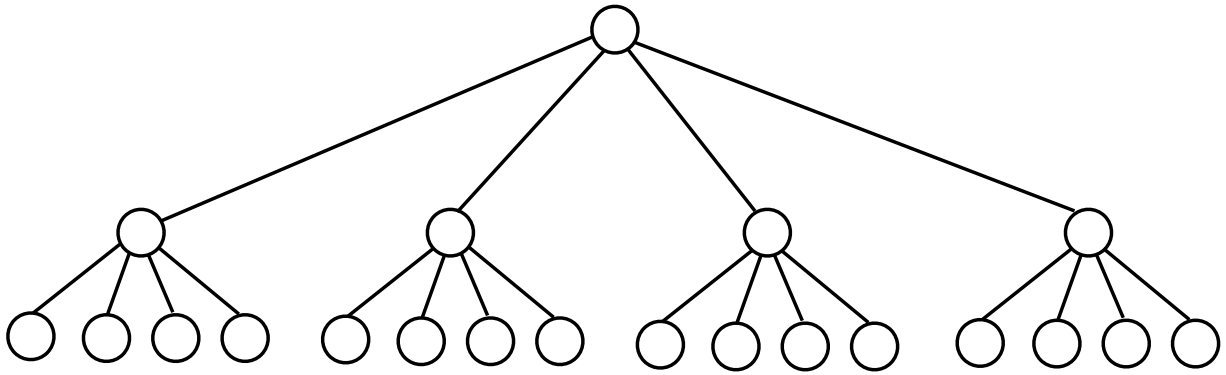
Theorem:
The amortized cost of a series of operations started from an empty splay tree is of order O(log N), where N is the number of all nodes involved in the operations.
Read the original splay tree paper for details.

# Balanced Search Trees (II)

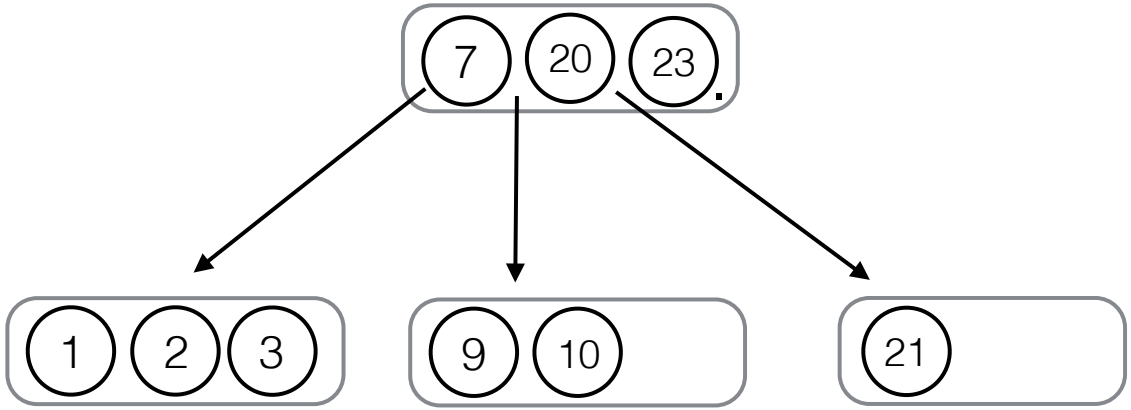
- Review of amortized analysis
- Red-black trees
- B+ trees
- Take-home messages

# Generalizing Balanced BSTs

- AVL trees and Splay trees are good for searching due to the balancing condition. But if we want fewer rotation operations when inserting and deleting:
  - Sacrifice a little searching cost
  - Relax balancing condition



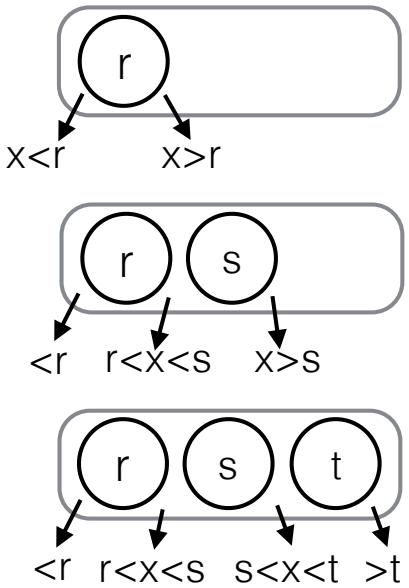
# M-Ary Search Trees

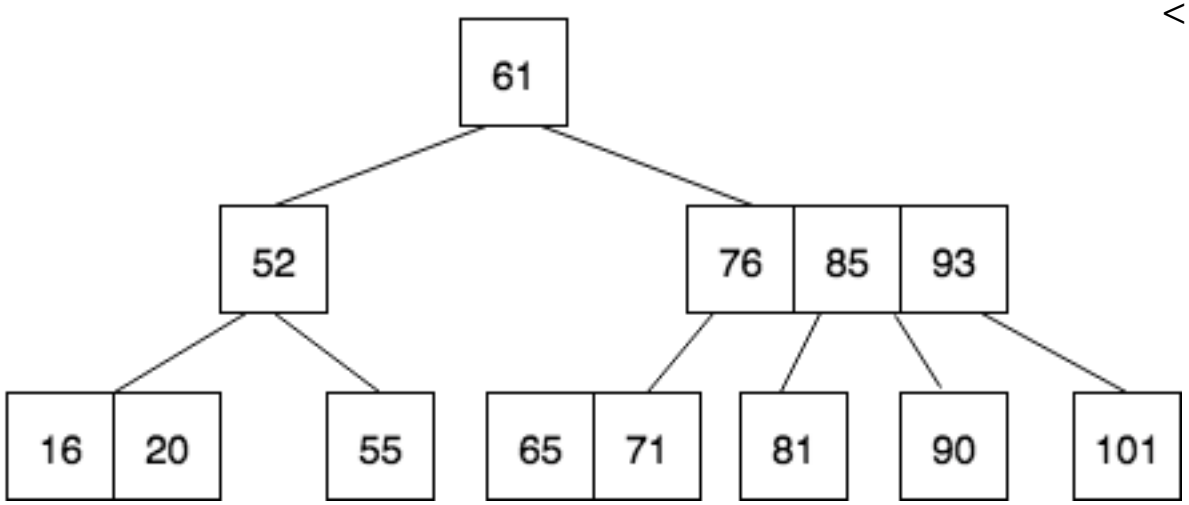



4-ary search tree:
Nodes have 1,2, or 3 data items and 0 to 4 children.

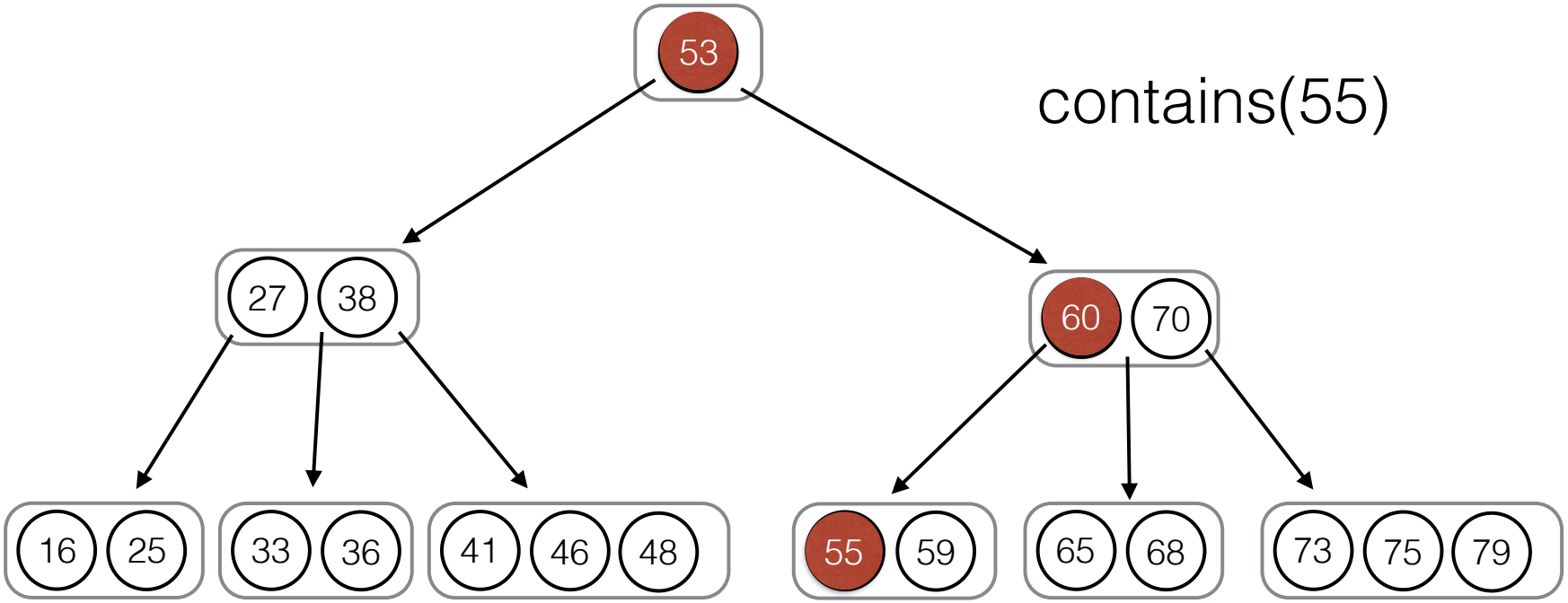
# 2-3-4 Trees

- A 2-3-4 tree is a balanced 4-Ary search tree.
- Three types of internal nodes:
- Balance condition:
  - All leaves have the same depth.

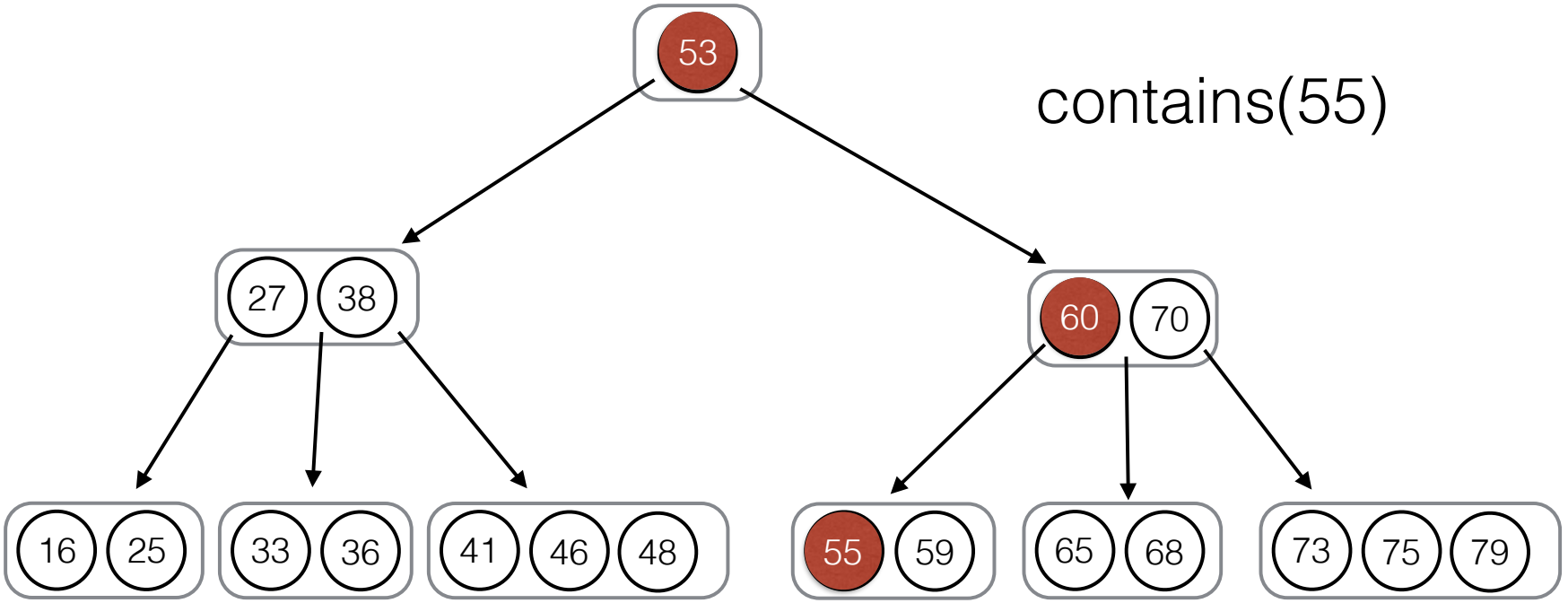


# Searching



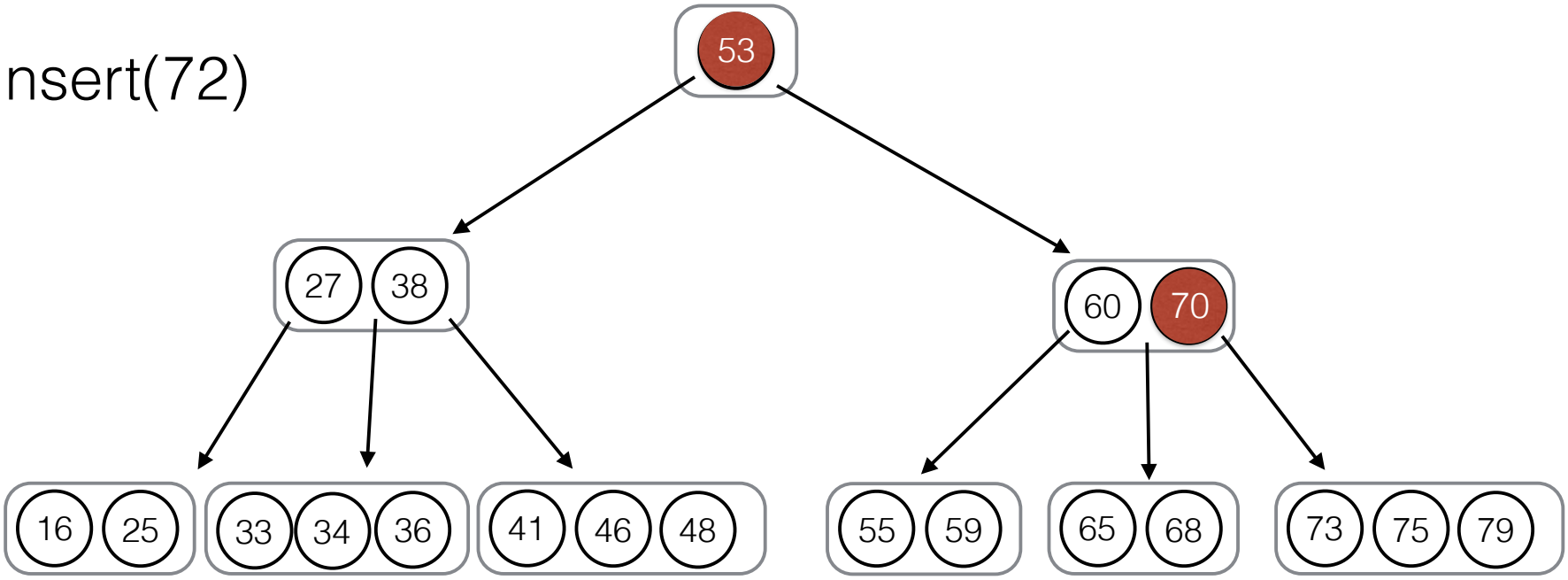
# Searching




Linear search in each node. O(3d) time cost.

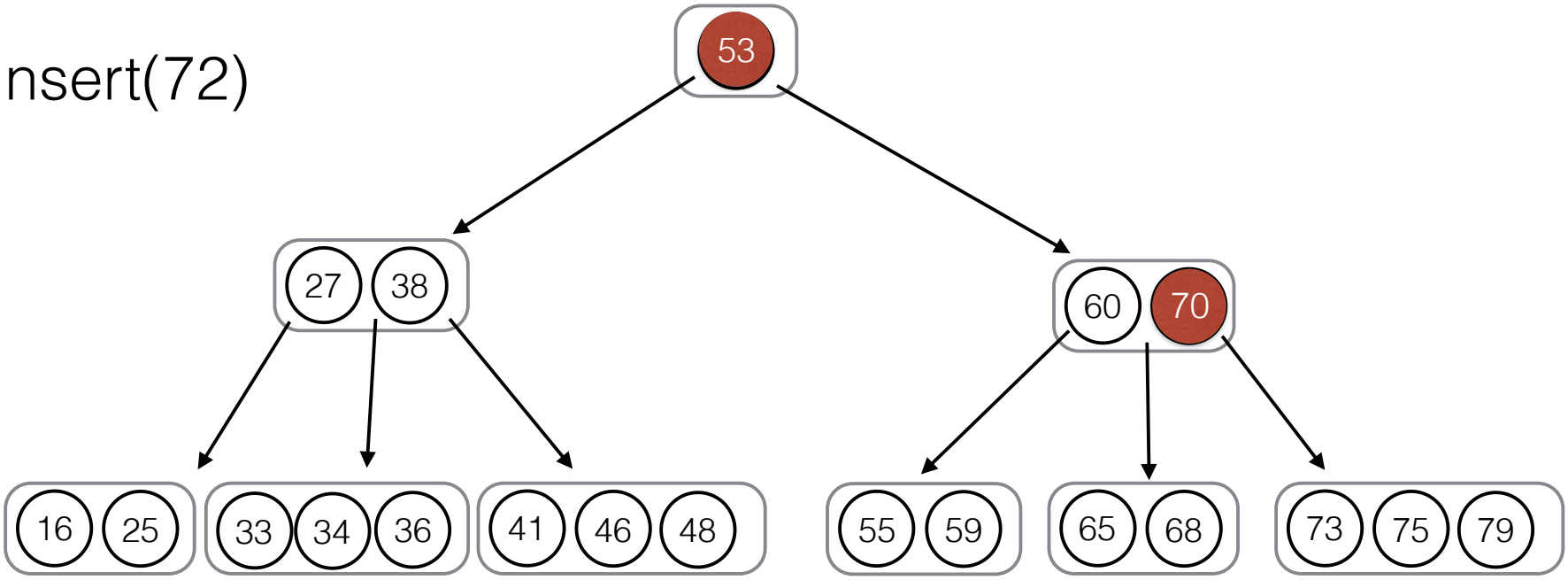
# Insertion

insert(72)



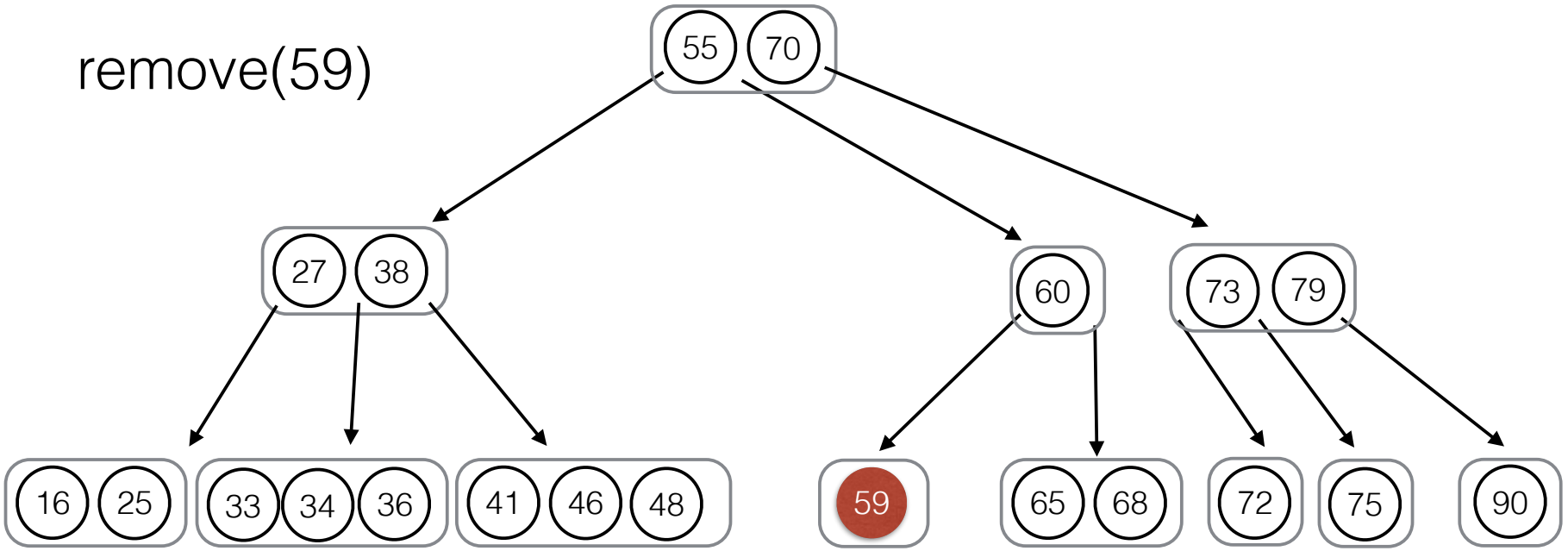
# Insertion

insert(72)




The insertion happens on the leaves.
When the leaf is full, splitting needs to be done.

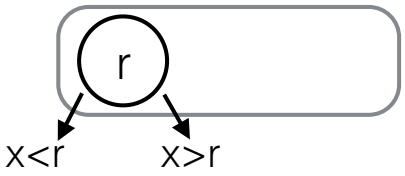
# Deletion

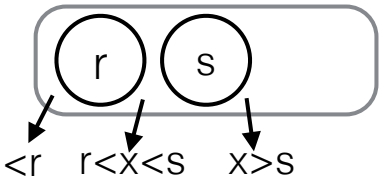


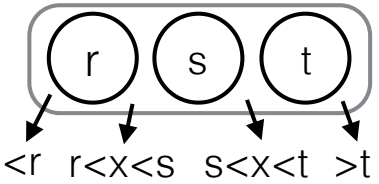

Deletions can make the nodes don't satisfy the minimum number of keys (e.g. 2).
Needs further manipulation (combine)
Can we make insertion and deletion easy with binary search tree?

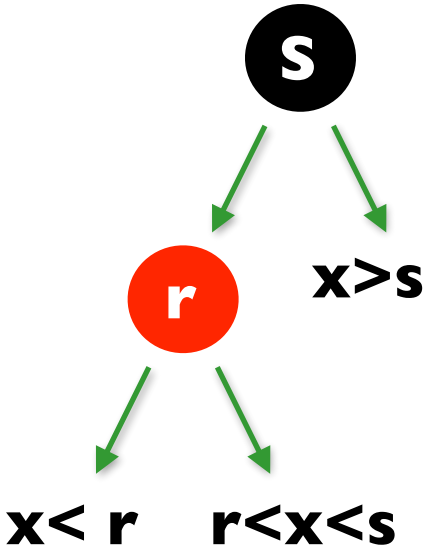
# Red-Black Trees

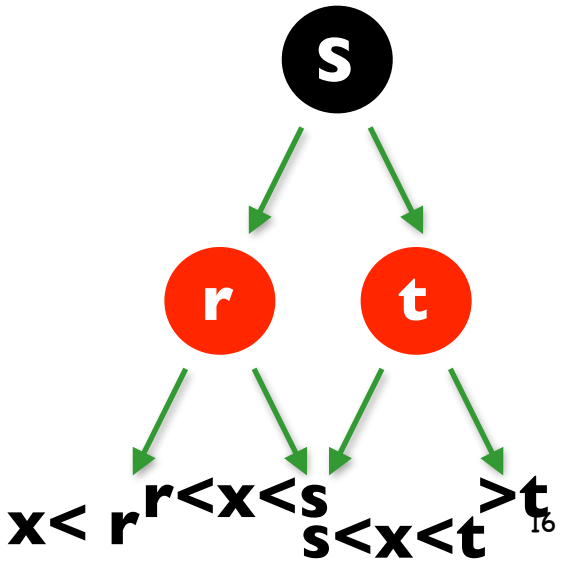
- Reduce 2-3-4 trees to BSTs:
  - The key is to transform 3- and 4- nodes into 2-nodes:



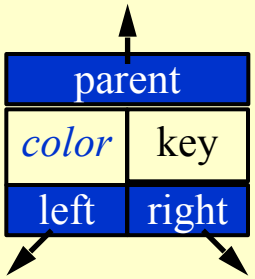
# Red-Black Trees

# Red-Black Trees

**Target :** Balanced binary search tree

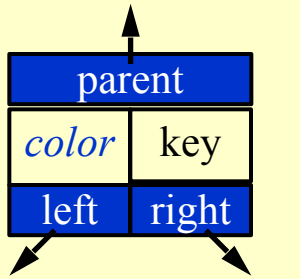
# Red-Black Trees

**Target :** Balanced binary search tree



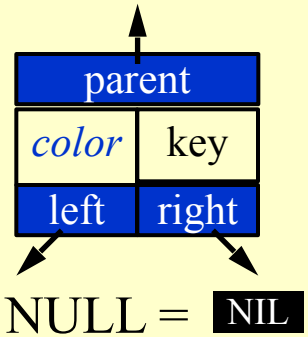
# Red-Black Trees

**Target :** Balanced binary search tree



# Red-Black Trees

**Target :** Balanced binary search tree




【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

(1) Every node is either **red** or **black**.

(2) The root is **black**.

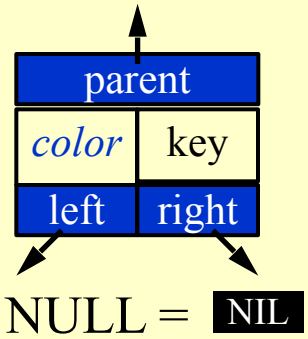
(3) Every leaf (NIL) is **black**.

(4) If a node is **red**, then both its children are **black**.

(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.

# Red-Black Trees

**Target :** Balanced binary search tree




【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

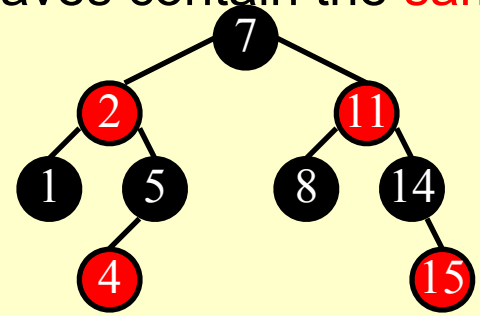
(1) Every node is either **red** or **black**.

(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

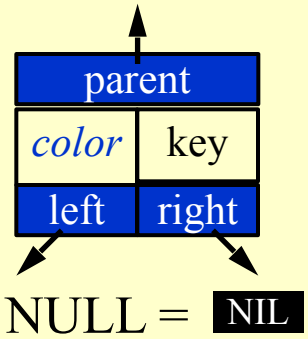
(4) If a node is **red**, then both its children are **black**.

(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.



# Red-Black Trees

**Target :** Balanced binary search tree




【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

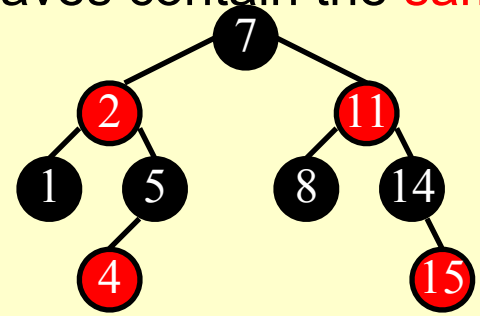
(1) Every node is either **red** or **black**.

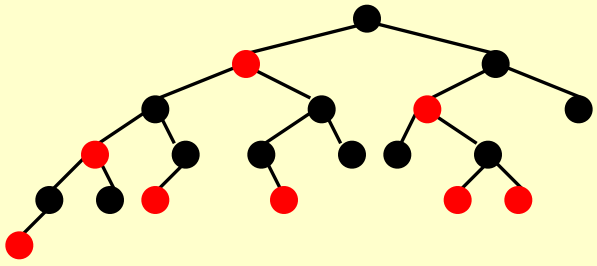
(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

(4) If a node is **red**, then both its children are **black**.

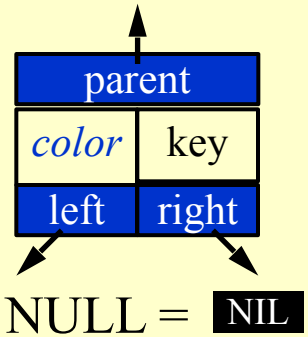
(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.



# Red-Black Trees

**Target :** Balanced binary search tree




【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

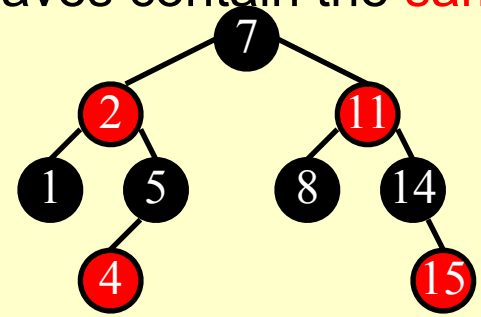
(1) Every node is either **red** or **black**.

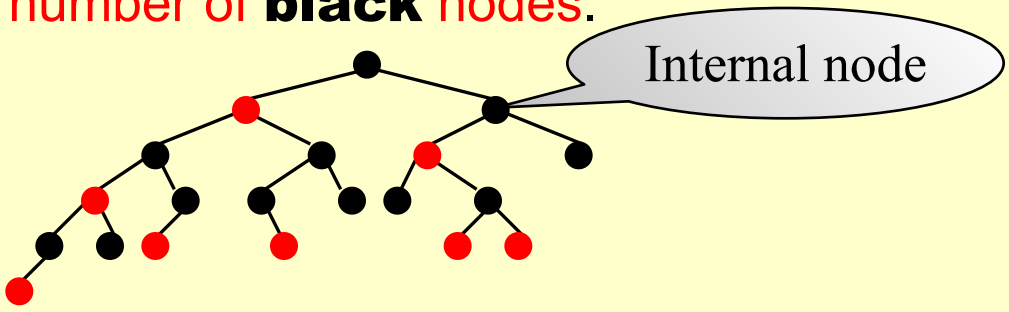
(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

(4) If a node is **red**, then both its children are **black**.

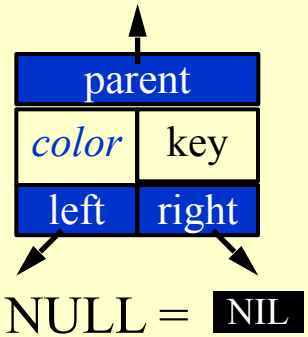
(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.



# Red-Black Trees

**Target :** Balanced binary search tree




【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

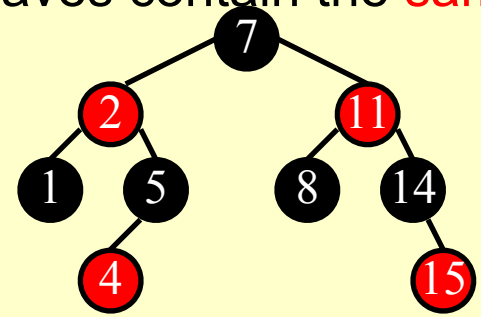
(1) Every node is either **red** or **black**.

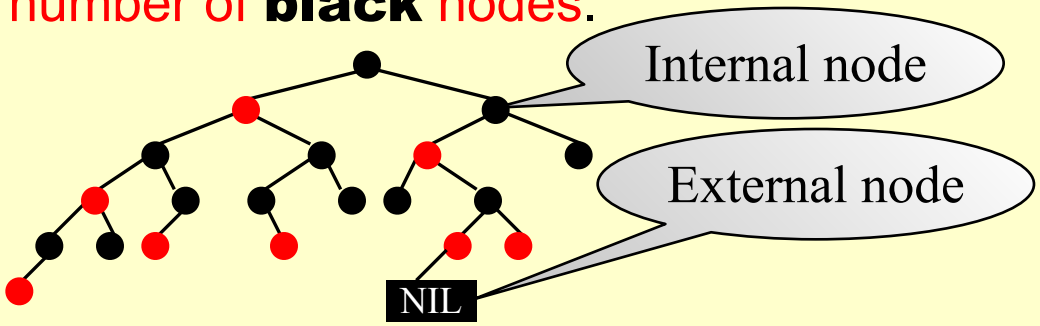
(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

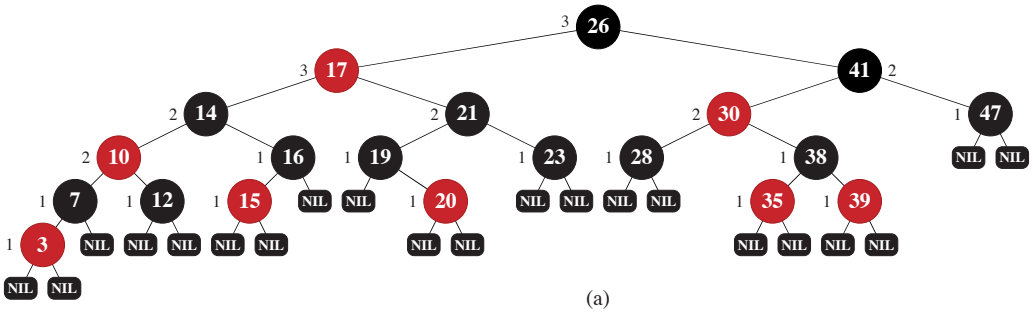
(4) If a node is **red**, then both its children are **black**.

(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.

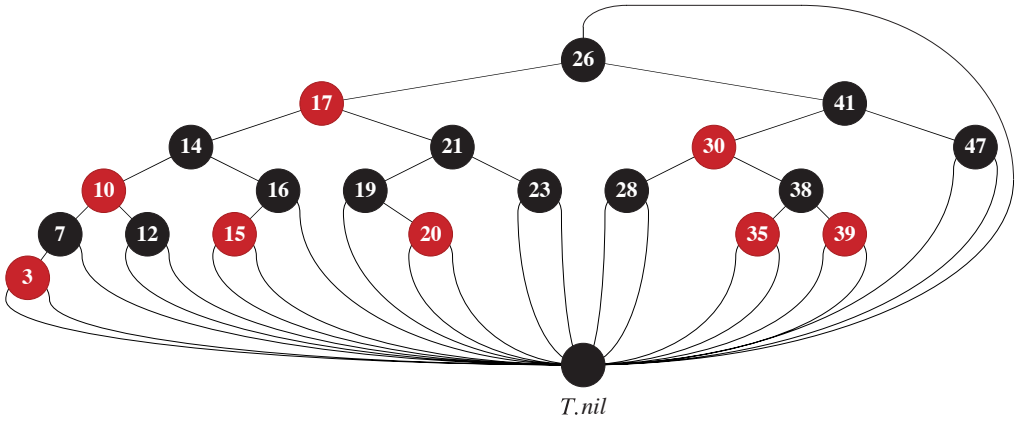


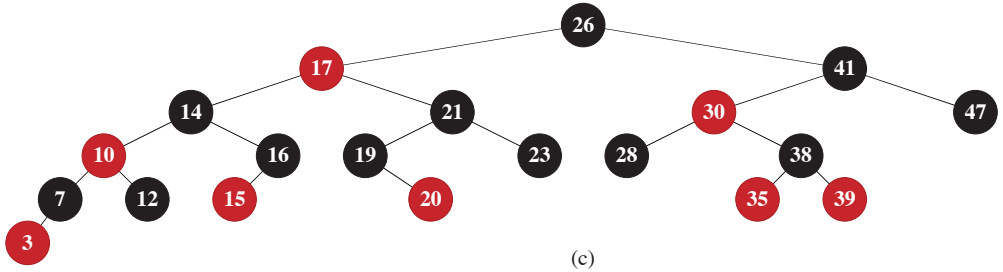
# Red-Black Trees




(a)




(b)




(c)

How balanced are red-black trees?

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf.

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Number of internal nodes in the subtree rooted at $x$

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh($child$) = ?

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh(*child*) = ? bh($x$) or bh($x$) – 1

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh($child$) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof($child$) ≥ $2^{bh(child)} - 1$

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh($child$) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof($child$) ≥ $2^{bh(child)} - 1$ ≥ $2^{bh(x)-1} - 1$

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh(*child*) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof(*child*) ≥ $2^{bh(child)} - 1$ ≥ $2^{bh(x)-1} - 1$

Hence sizeof($x$) = $1 + 2$sizeof(*child*) ≥ $2^{bh(x)} - 1$

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh(*child*) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof(*child*) ≥ $2^{bh(child)} - 1$ ≥ $2^{bh(x)-1} - 1$

Hence sizeof($x$) = $1 + 2\text{sizeof}(child) \ge 2^{bh(x)} - 1$ ✔

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh($child$) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof($child$) ≥ $2^{bh(child)} - 1$ ≥ $2^{bh(x)-1} - 1$

Hence sizeof($x$) = 1 + 2sizeof($child$) ≥ $2^{bh(x)} - 1$ ✔

② bh($Tree$) ≥ $h(Tree) / 2$ ?

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh(*child*) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof(*child*) ≥ $2^{bh(child)} - 1$ ≥ $2^{bh(x)-1} - 1$

Hence sizeof($x$) = 1 + 2sizeof(*child*) ≥ $2^{bh(x)} - 1$ ✔

② bh(*Tree*) ≥ $h(Tree) / 2$ ?

**Discussion 2:** Please finish the proof.

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh(*child*) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof(*child*) ≥ $2^{bh(child)} - 1$ ≥ $2^{bh(x)-1} - 1$

Hence sizeof($x$) = 1 + 2sizeof(*child*) ≥ $2^{bh(x)} - 1$ ✔

② bh(*Tree*) ≥ $h(Tree) / 2$ ?

Since for every red node, both of its children must be black, hence on any simple path from *root* to a leaf, at least half the nodes (*root* not included) must be black.

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a leaf. bh(Tree) = bh(root).

【Lemma】 A red-black tree with $N$ internal nodes has height at most $2\ln(N+1)$.

Proof: ① For any node $x$, sizeof($x$) ≥ $2^{bh(x)} - 1$. Prove by induction.

If $h(x) = 0$, $x$ is NULL ➡ sizeof($x$) = $2^0 - 1 = 0$ ✔

Suppose it is true for all $x$ with $h(x) \le k$.

For $x$ with $h(x) = k + 1$, bh(*child*) = ? bh($x$) or bh($x$) – 1

Since $h(child) \le k$, sizeof(*child*) ≥ $2^{bh(child)} - 1$ ≥ $2^{bh(x)-1} - 1$

Hence sizeof($x$) = 1 + 2sizeof(*child*) ≥ $2^{bh(x)} - 1$ ✔

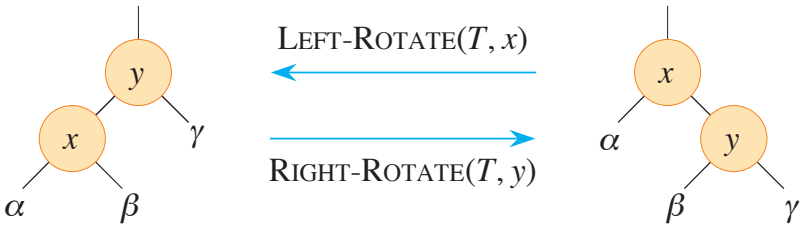
② bh(*Tree*) ≥ $h$(*Tree*) / 2 ?

Since for every red node, both of its children must be black, hence on any simple path from *root* to a leaf, at least half the nodes (*root* not included) must be black.

$$\text{Sizeof}(root) = N \ge 2^{bh(Tree)} - 1 \ge 2^{h/2} - 1$$

# Tree Insertion and Deletion

- Similar with AVL and splay trees, the rotations are usually required when insertion and deletion lead to the violation of tree properties.



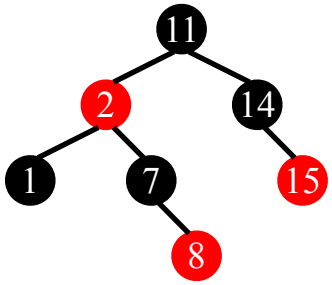

```
LEFT-ROTATE(T, x)
1   y = x.right
2   x.right = y.left      // turn y's left subtree into x's right subtree
3   if y.left ≠ T.nil     // if y's left subtree is not empty ...
4       y.left.p = x      // ... then x becomes the parent of the subtree's root
5   y.p = x.p             // x's parent becomes y's parent
6   if x.p == T.nil       // if x was the root ...
7       T.root = y        // ... then y becomes the root
8   elseif x == x.p.left  // otherwise, if x was a left child ...
9       x.p.left = y      // ... then y becomes a left child
10  else x.p.right = y    // otherwise, x was a right child, and now y is
11  y.left = x            // make x become y's left child
12  x.p = y
```

Need to reduce the number of rotations

# Insertion

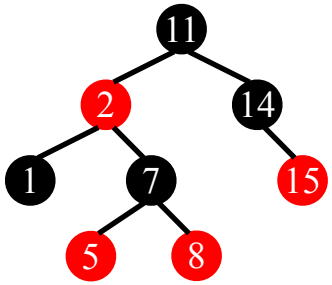
always insert the new node as a red node on the bottom.

# Insertion



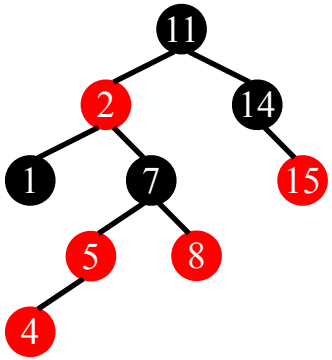

always insert the new node as a red node on the bottom.

# Insertion



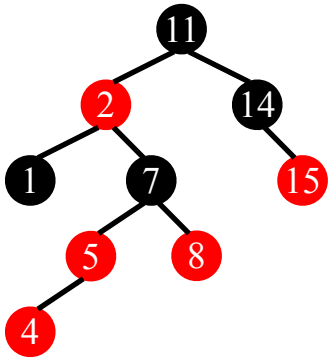

always insert the new node as a red node on the bottom.

# Insertion




always insert the new node as a red node on the bottom.

# Insertion




always insert the new node as a red node on the bottom.

【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

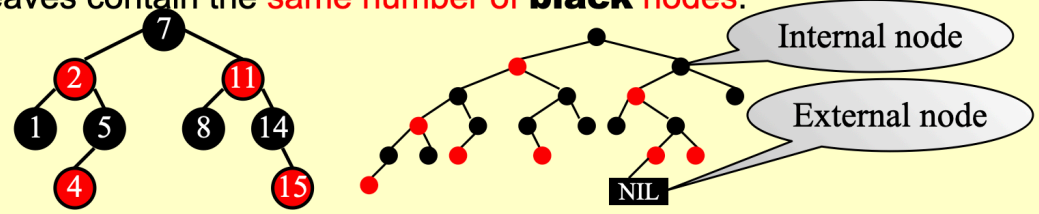
(1) Every node is either **red** or **black**.

(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

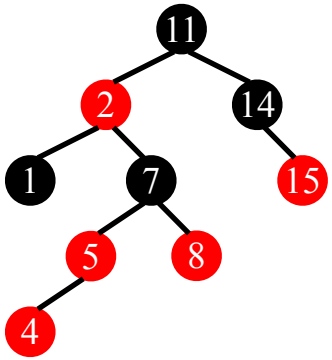
(4) If a node is **red**, then both its children are **black**.

(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.




What properties can be violated?

# Insertion




always insert the new node as a red node on the bottom.

【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

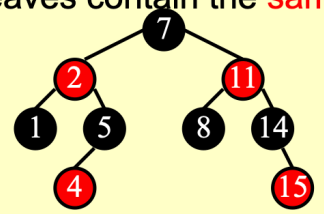
(1) Every node is either **red** or **black**.

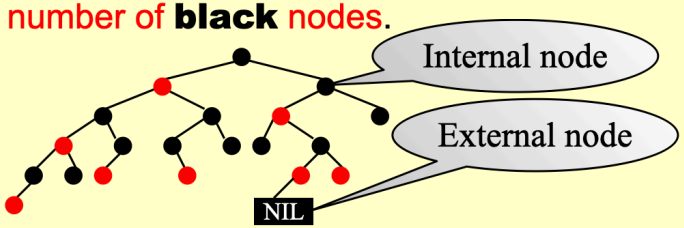
(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

(4) If a node is **red**, then both its children are **black**.

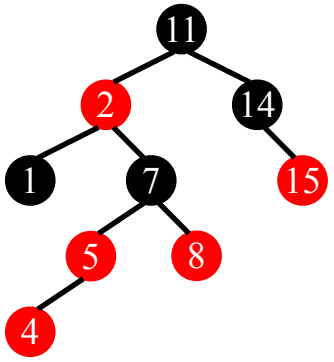
(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.




What properties can be violated?

# Insertion




always insert the new node as a red node on the bottom.

【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

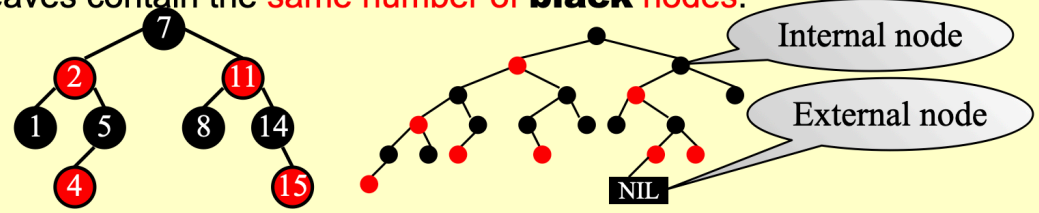
(1) Every node is either **red** or **black**.

(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

(4) If a node is **red**, then both its children are **black**.

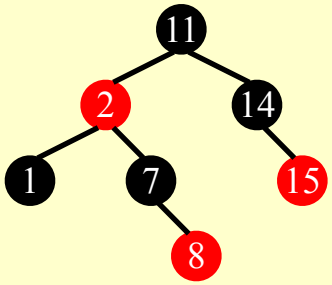
(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.



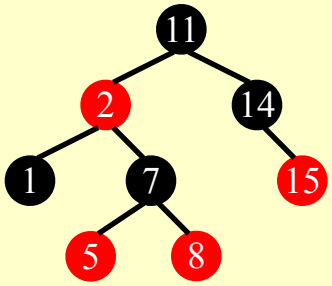

What properties can be violated?

Sketch of the idea: Insert & color red

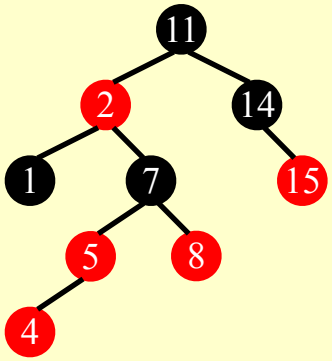
Sketch of the idea: Insert & color red



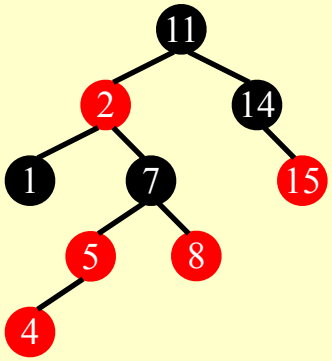
Sketch of the idea: Insert & color red



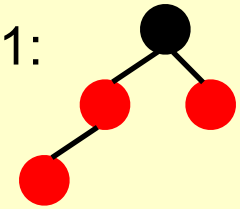
Sketch of the idea: Insert & color red



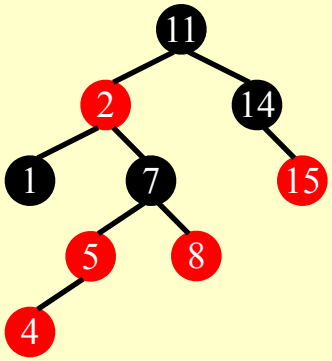
Sketch of the idea: Insert & color red



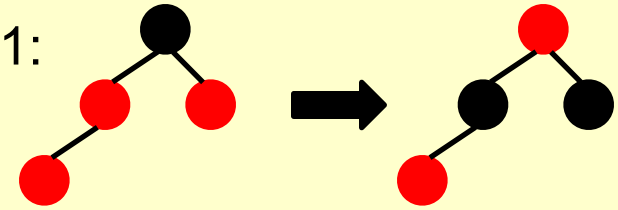

Case 1:



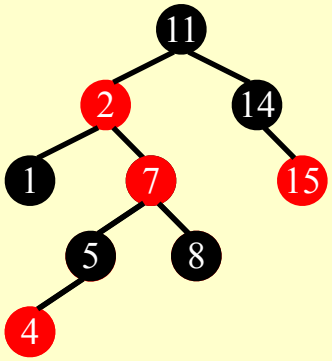
Sketch of the idea: Insert & color red



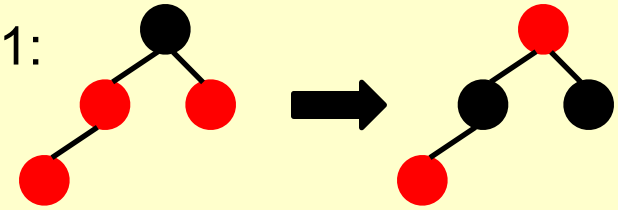

Case 1:



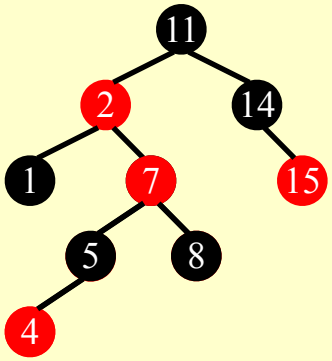
Sketch of the idea: Insert & color red



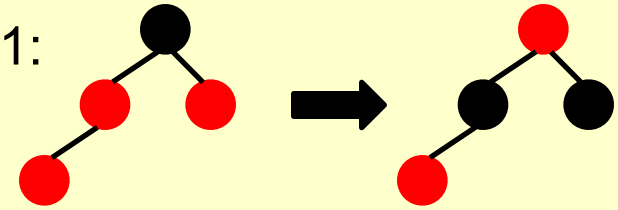

Case 1:



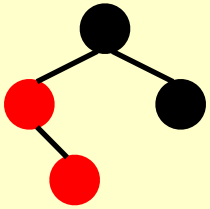
Sketch of the idea: Insert & color red



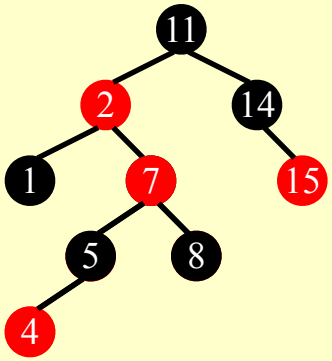

Case 1:



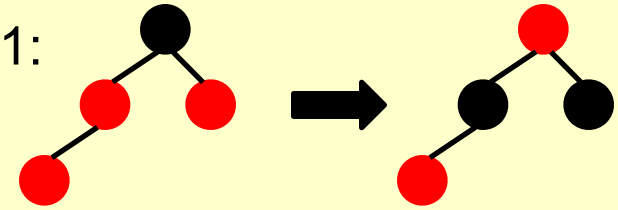
Case 2:



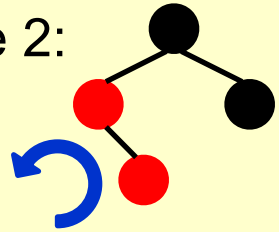
Sketch of the idea: Insert & color red



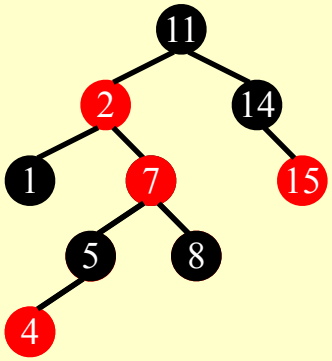

Case 1:



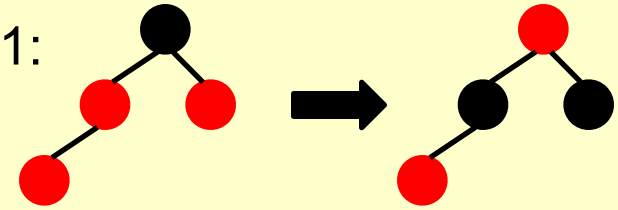
Case 2:



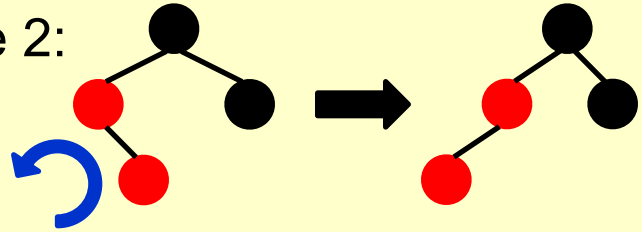
Sketch of the idea: Insert & color red



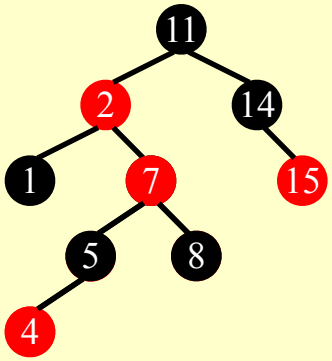

Case 1:

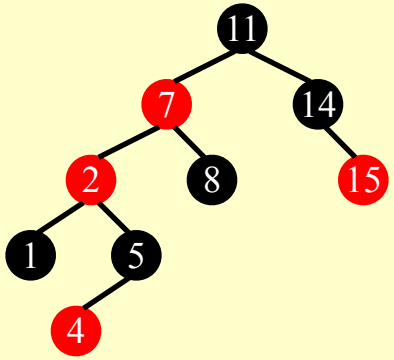


Case 2:

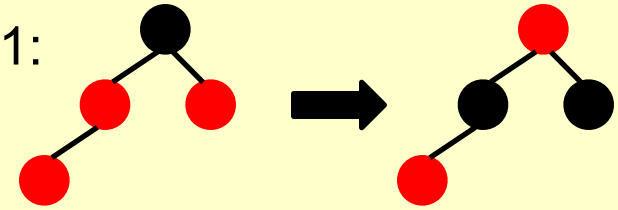


Sketch of the idea: Insert & color red

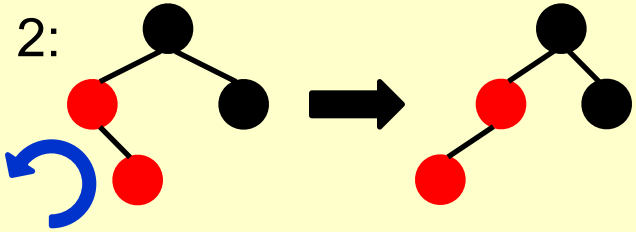


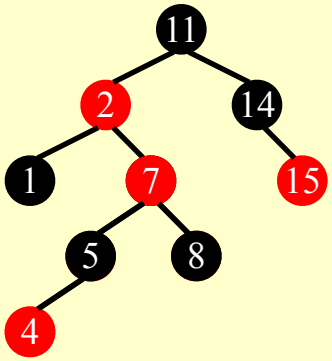

Case 1:

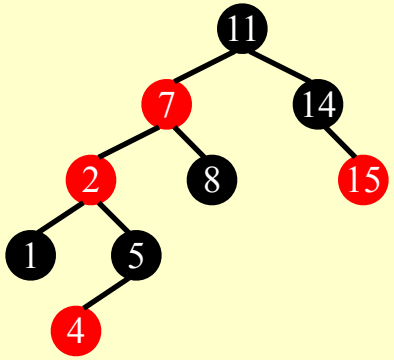


Case 2:

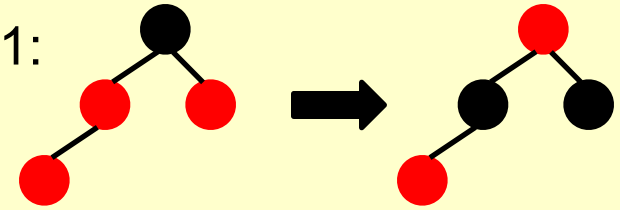


Sketch of the idea: Insert & color red

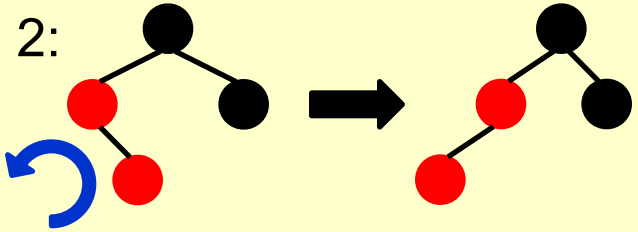


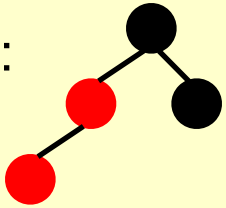

Case 1:



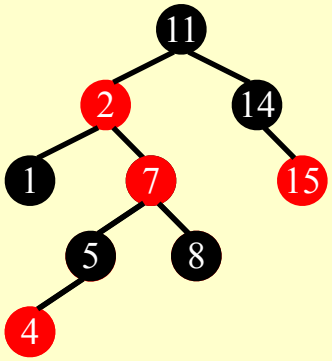
Case 2:

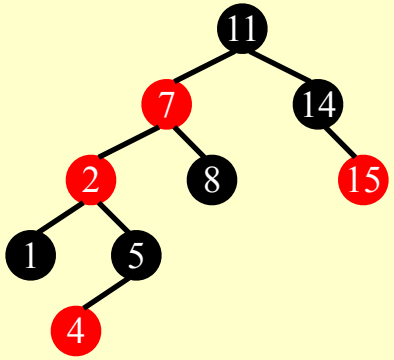


Case 3:

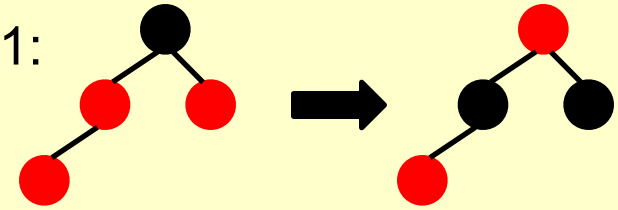


Sketch of the idea: Insert & color red

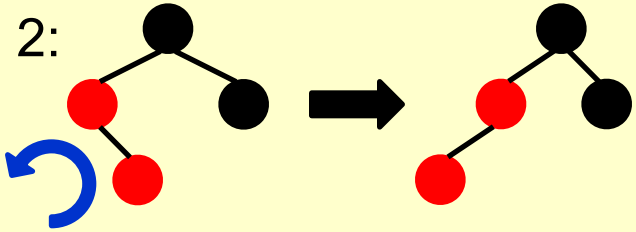


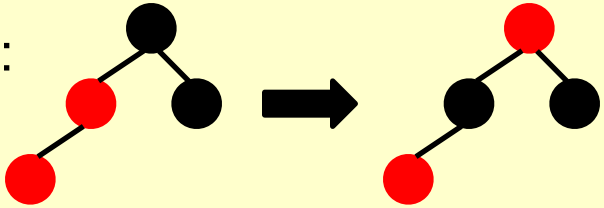

Case 1:



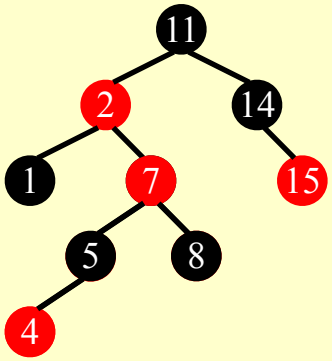
Case 2:

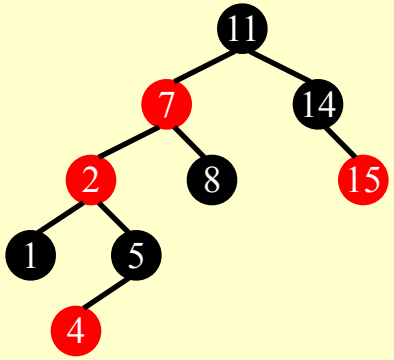


Case 3:

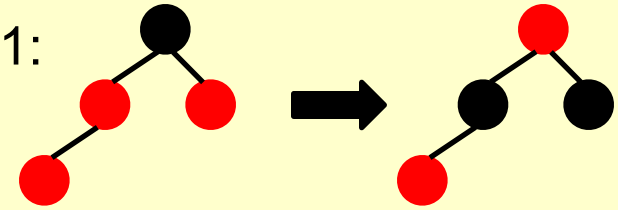


Sketch of the idea: Insert & color red

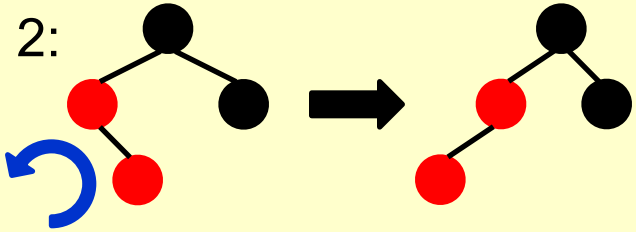


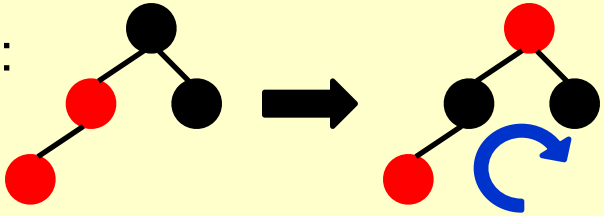

Case 1:



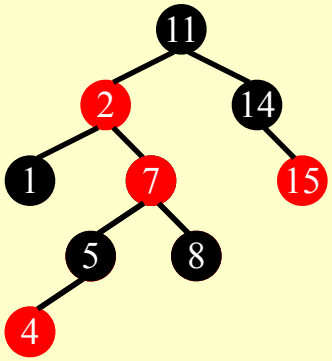
Case 2:

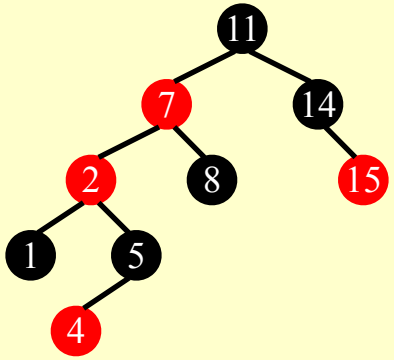


Case 3:

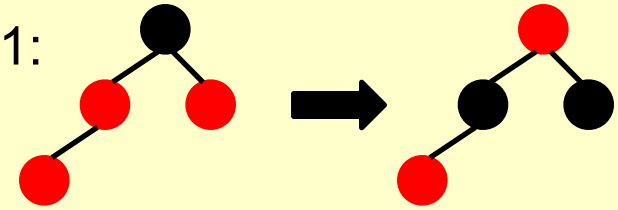


Sketch of the idea: Insert & color red

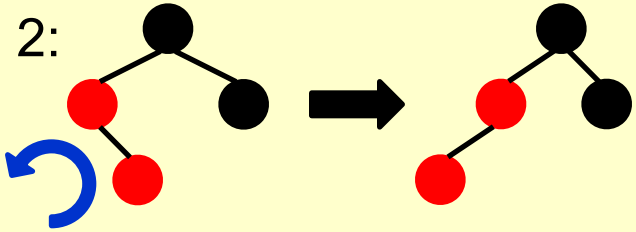


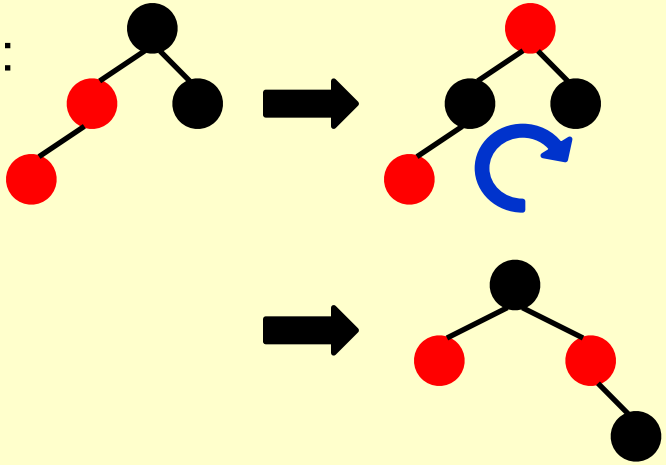

Case 1:



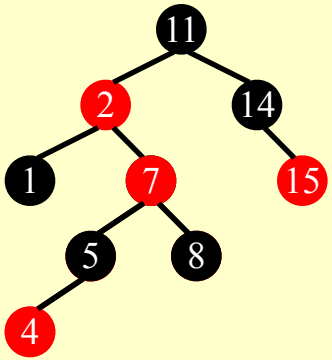
Case 2:

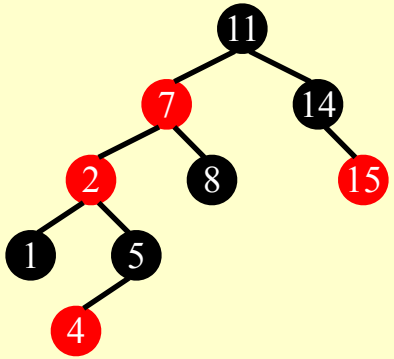


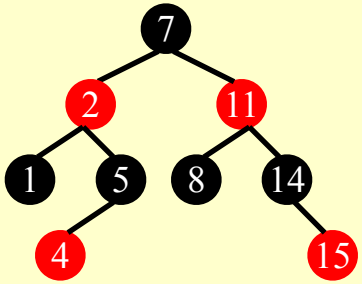
Case 3:



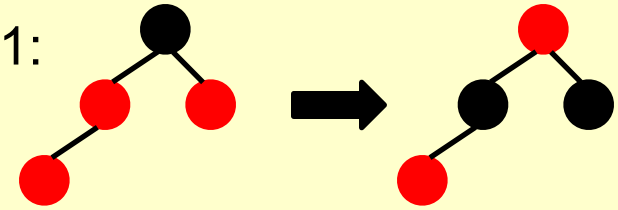
Sketch of the idea: Insert & color red



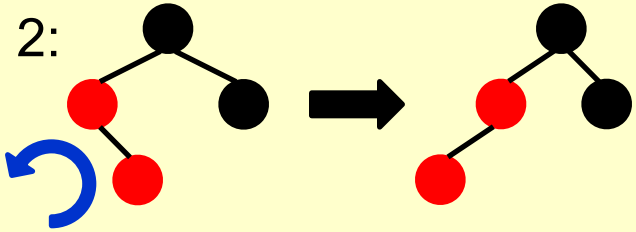



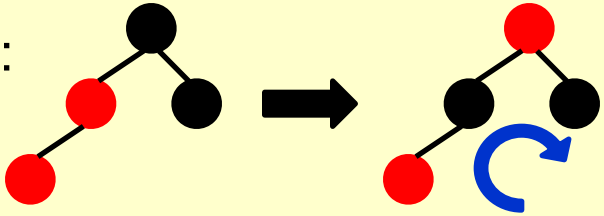

Case 1:

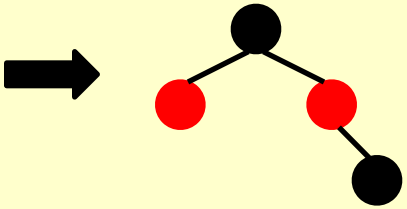


Case 2:



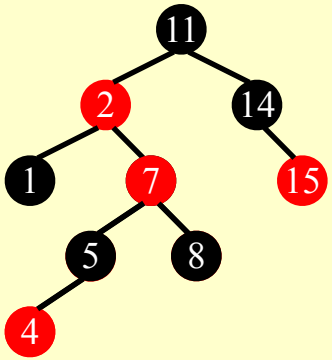
Case 3:

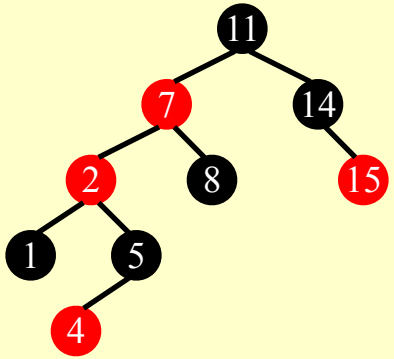


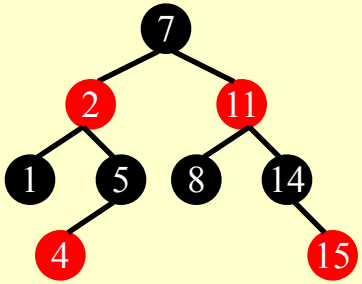


Sketch of the idea: Insert & color red

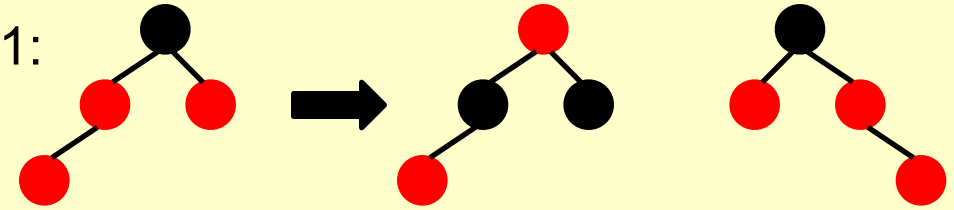
Symmetric



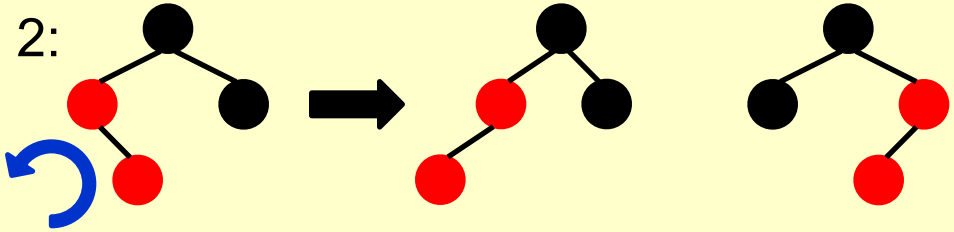



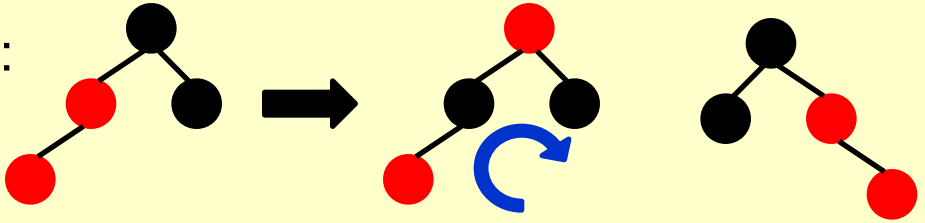

Case 1:

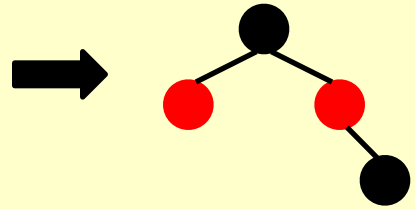


Case 2:



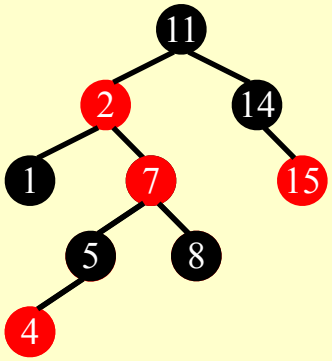
Case 3:



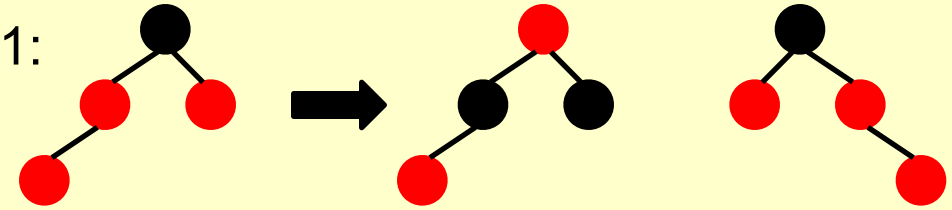


Sketch of the idea: Insert & color red

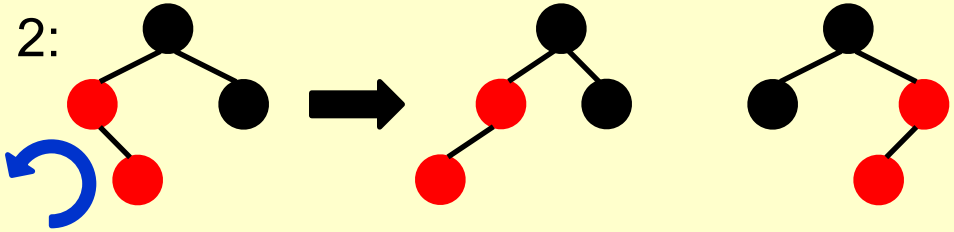
Symmetric

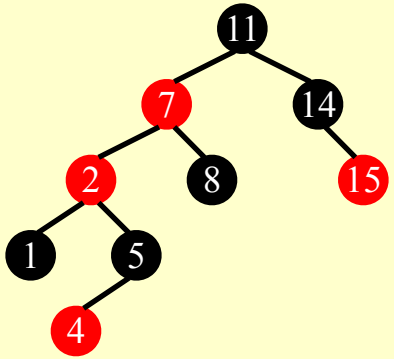



Case 1:

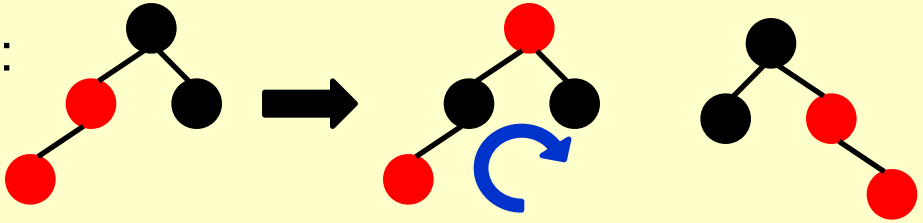


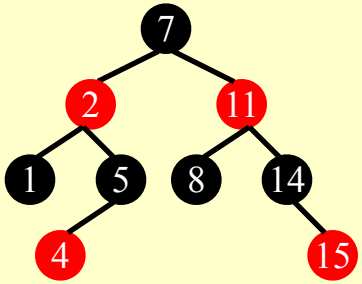
Case 2:

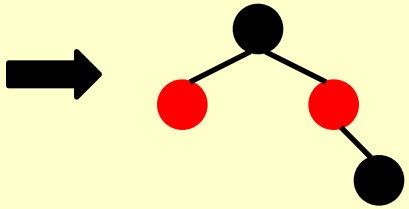





Case 3:





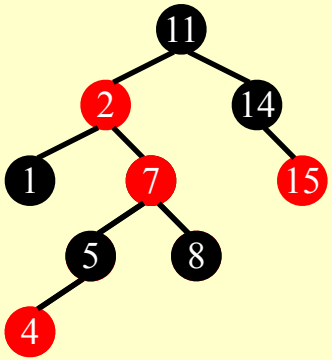



$T = O(h) = O(\ln N)$

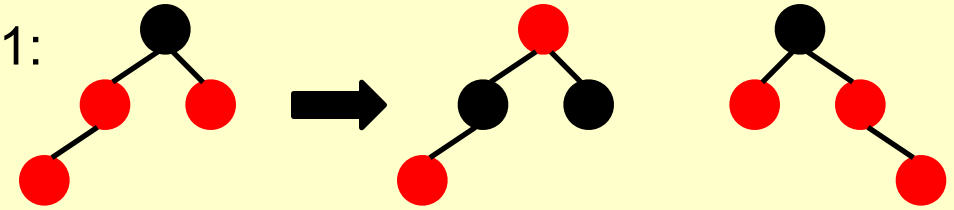
Only case 1 can repeat.
No more than 2 rotations.

Symmetric

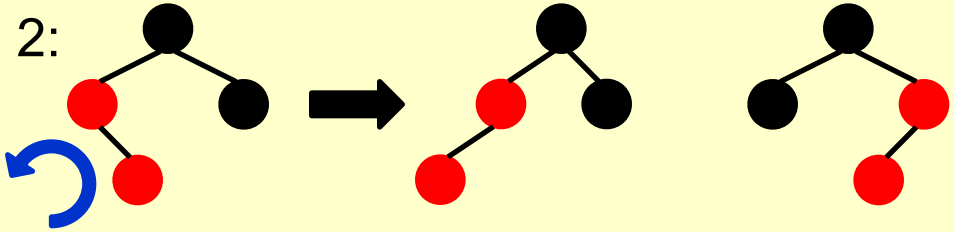
Sketch of the idea: Insert & color red

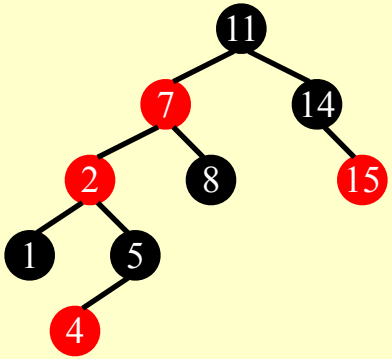



Case 1:

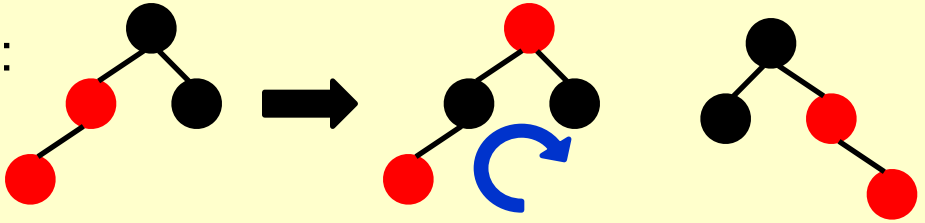


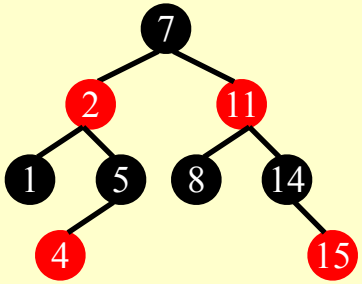
Case 2:

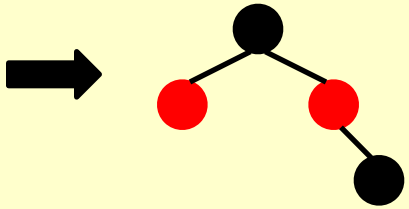





Case 3:



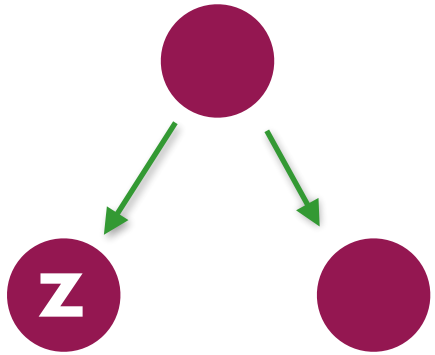


$T = O(h) = O(\ln N)$

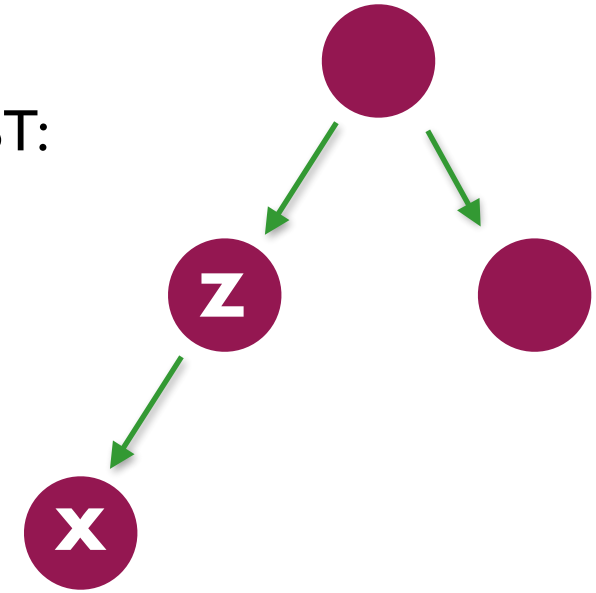
# Deletion

- Deletion in normal BST:



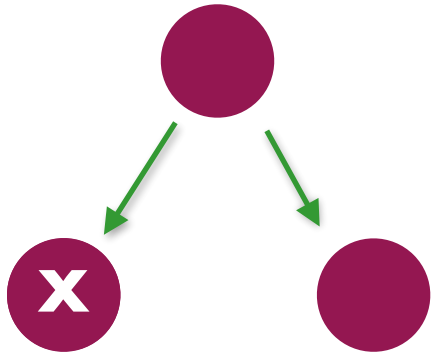
# Deletion

- Deletion in normal BST:



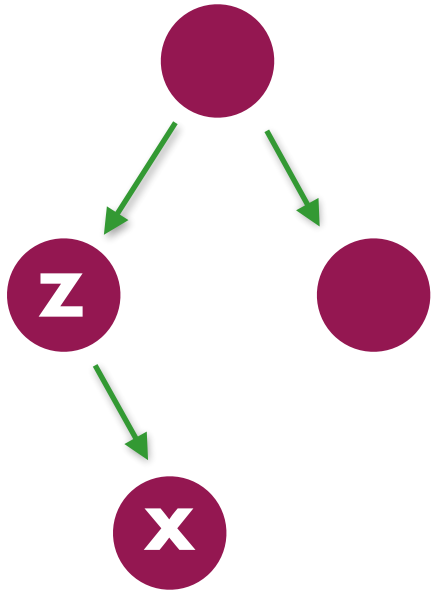
# Deletion

- Deletion in normal BST:



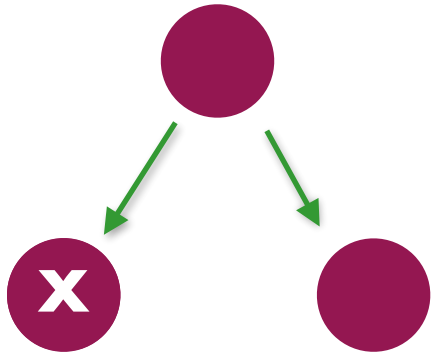
# Deletion

- Deletion in normal BST:



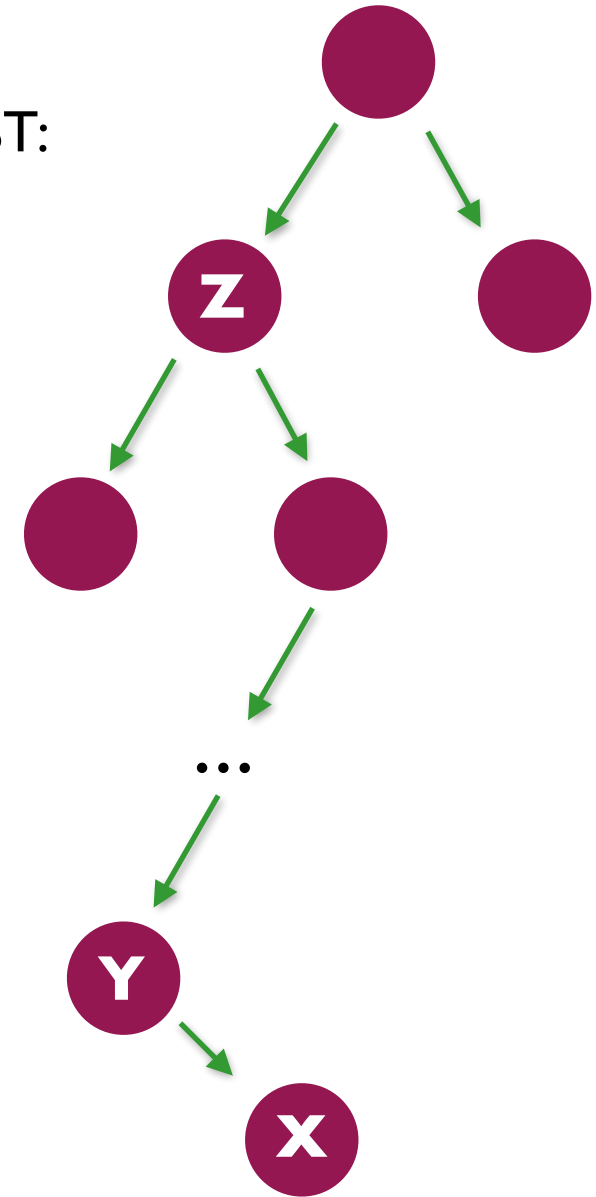
# Deletion

- Deletion in normal BST:



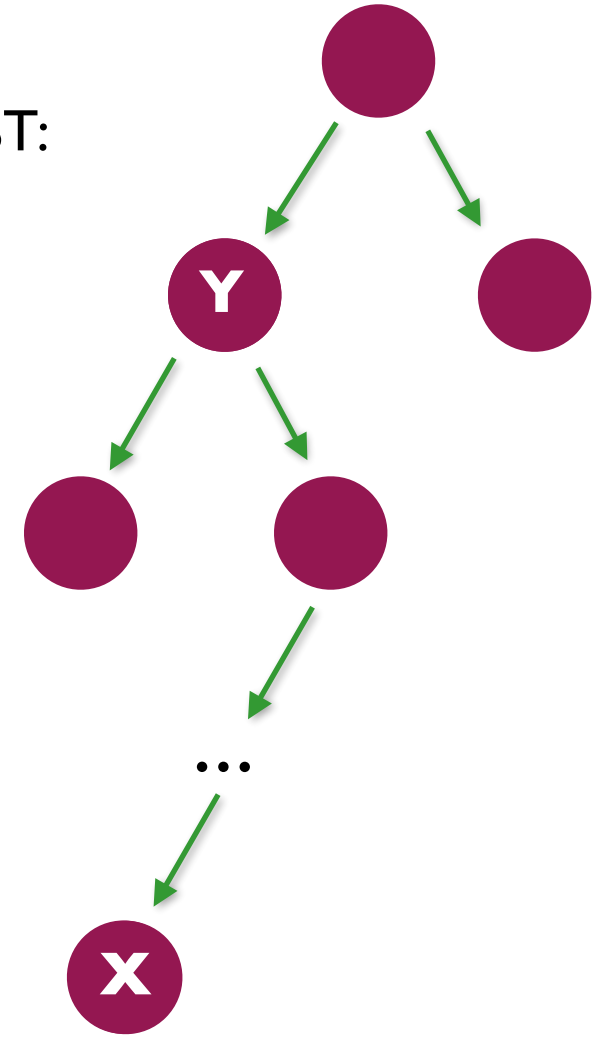
# Deletion

- Deletion in normal BST:



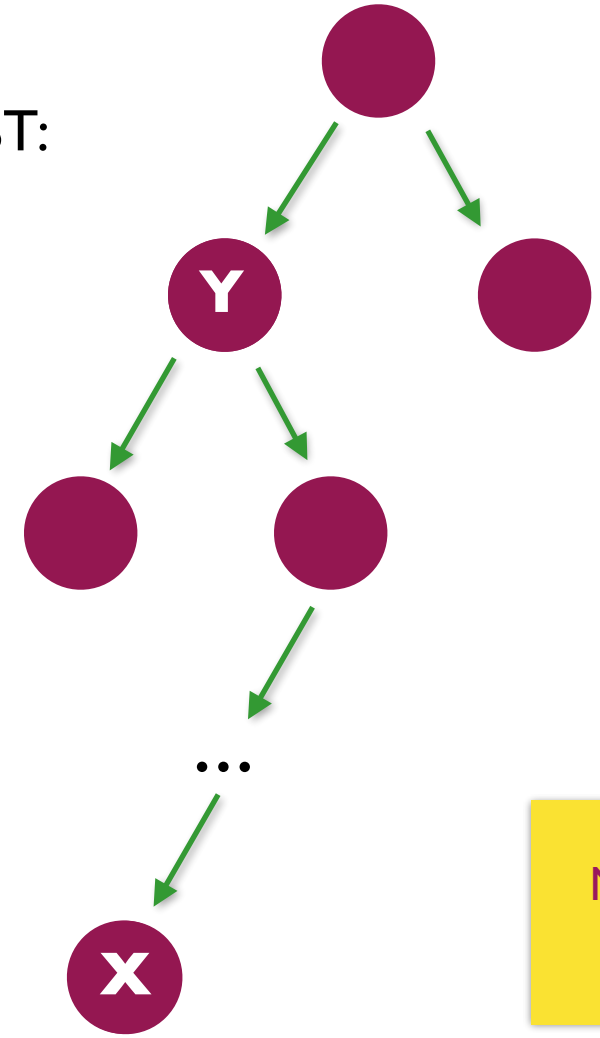
# Deletion

- Deletion in normal BST:



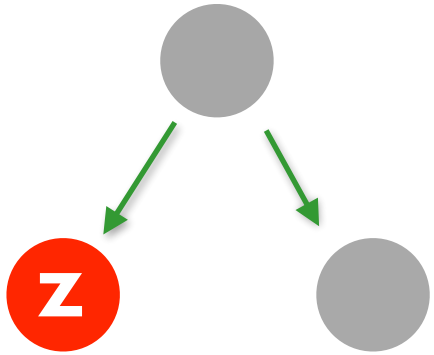
# Deletion

- Deletion in normal BST:

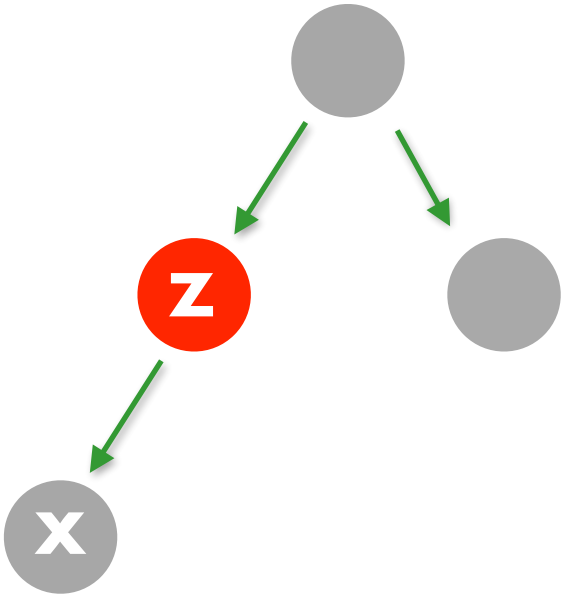



Now we need to consider color properties.

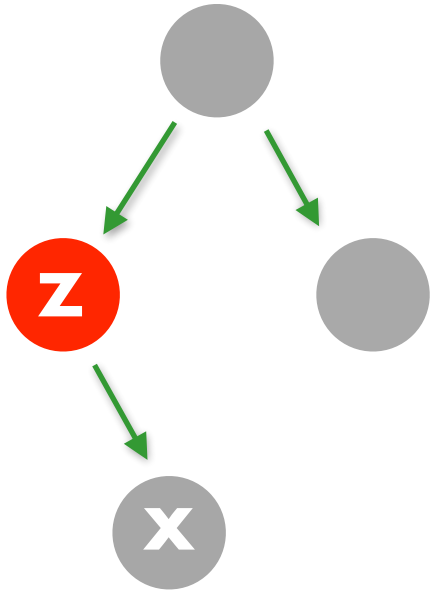
# Deletion



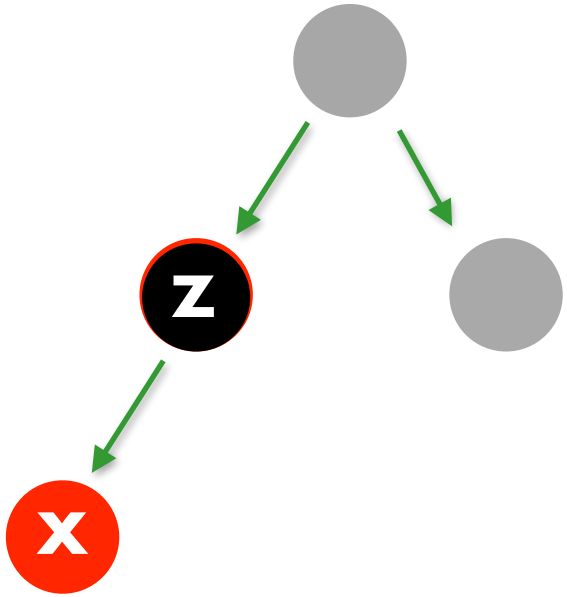
# Deletion



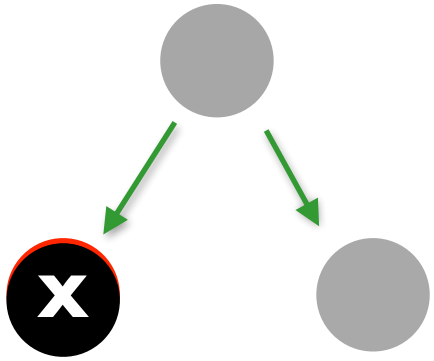
# Deletion



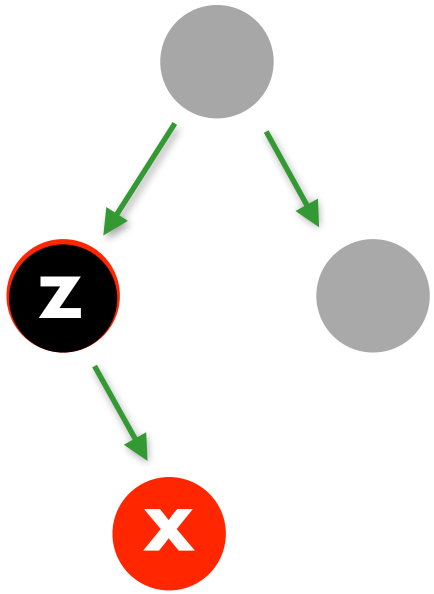
# Deletion



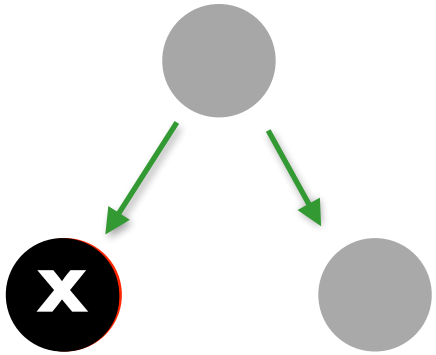
# Deletion



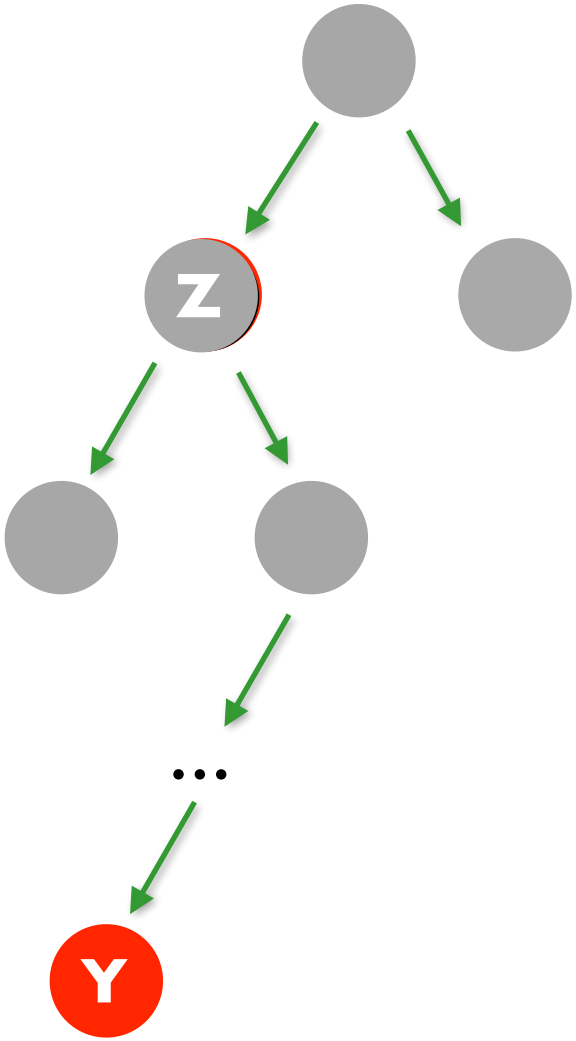
# Deletion



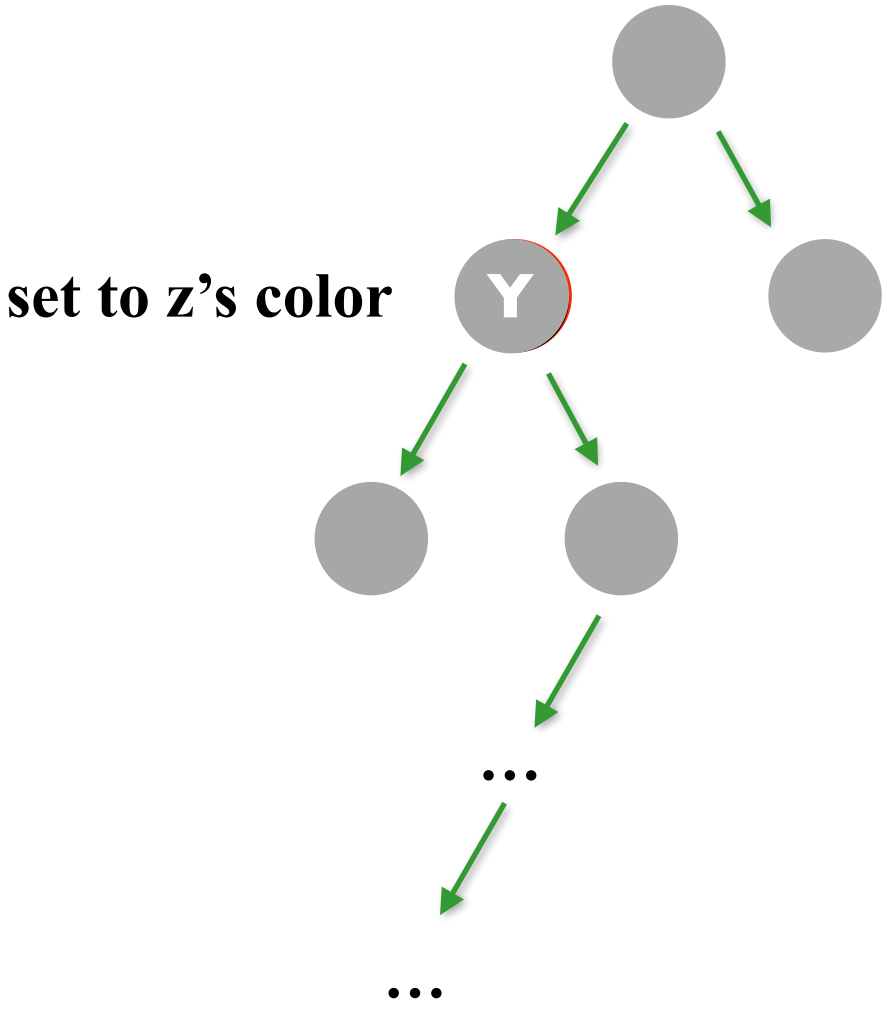
# Deletion



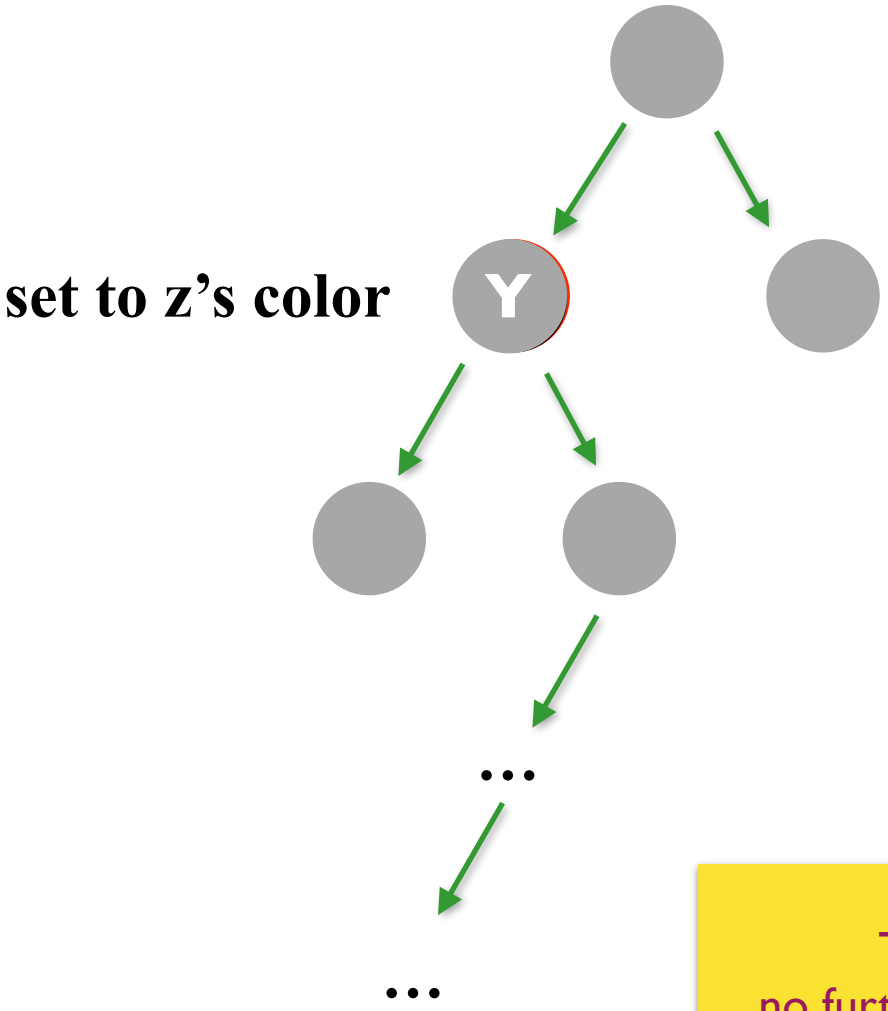
# Deletion



# Deletion

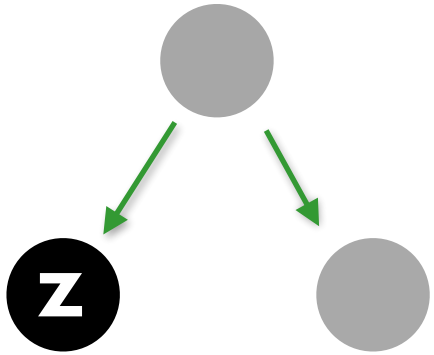


# Deletion

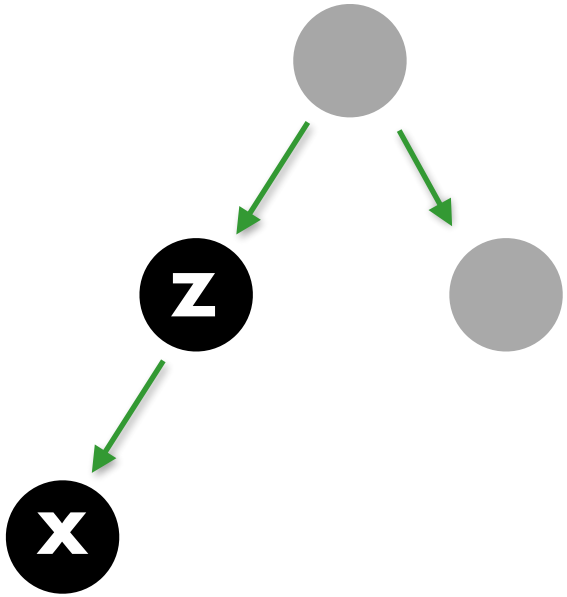



These are cases that
no further adjustment is needed.

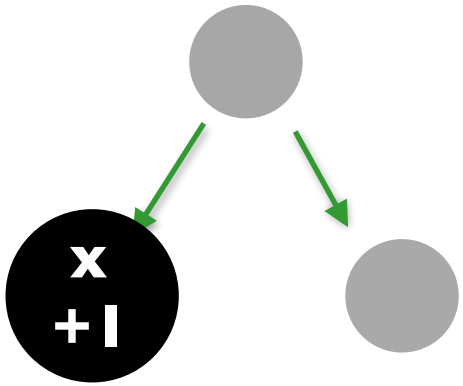
# Deletion



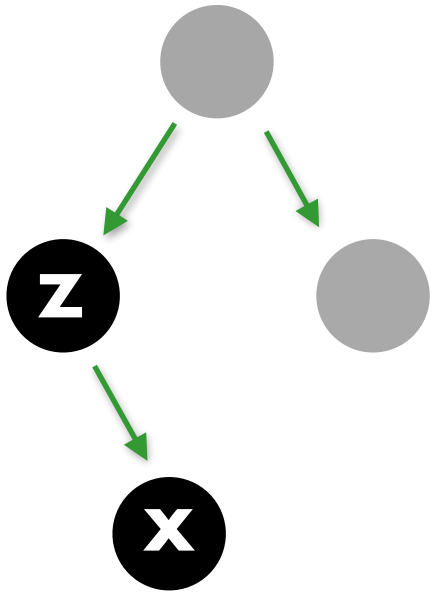
# Deletion



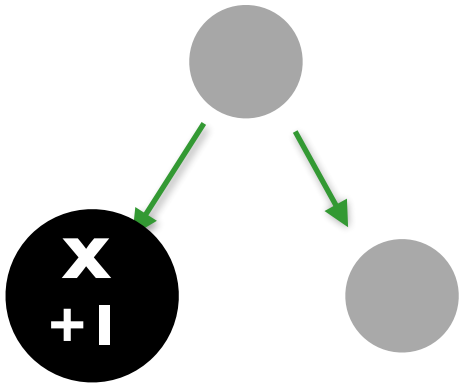
# Deletion



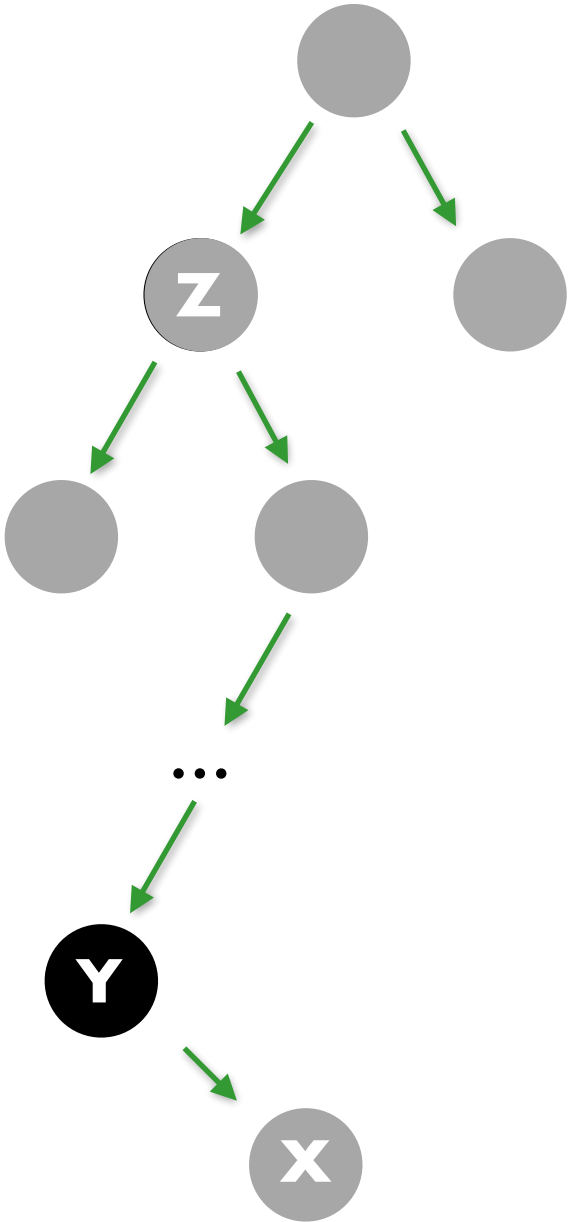
# Deletion



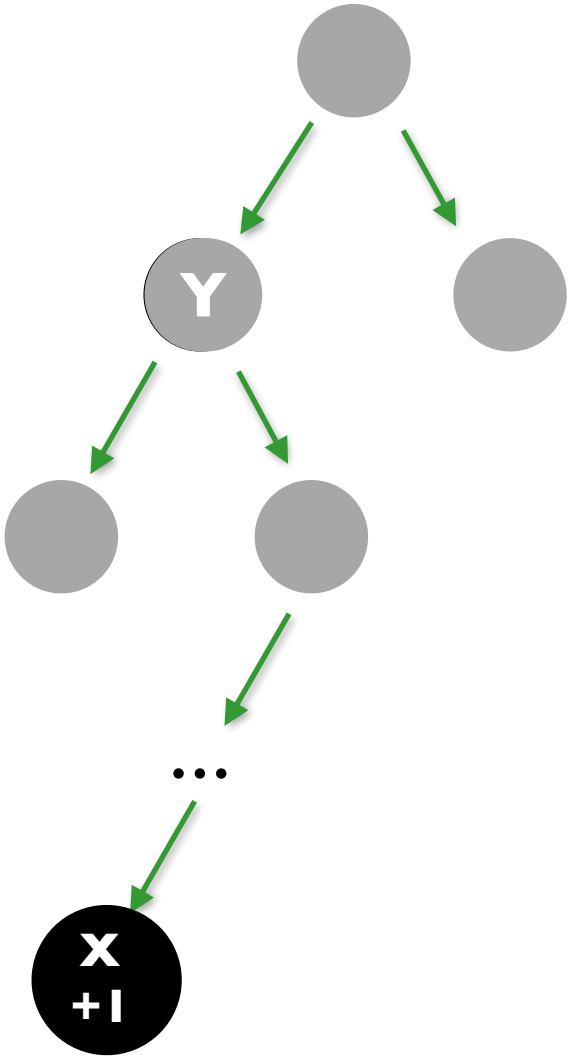
# Deletion



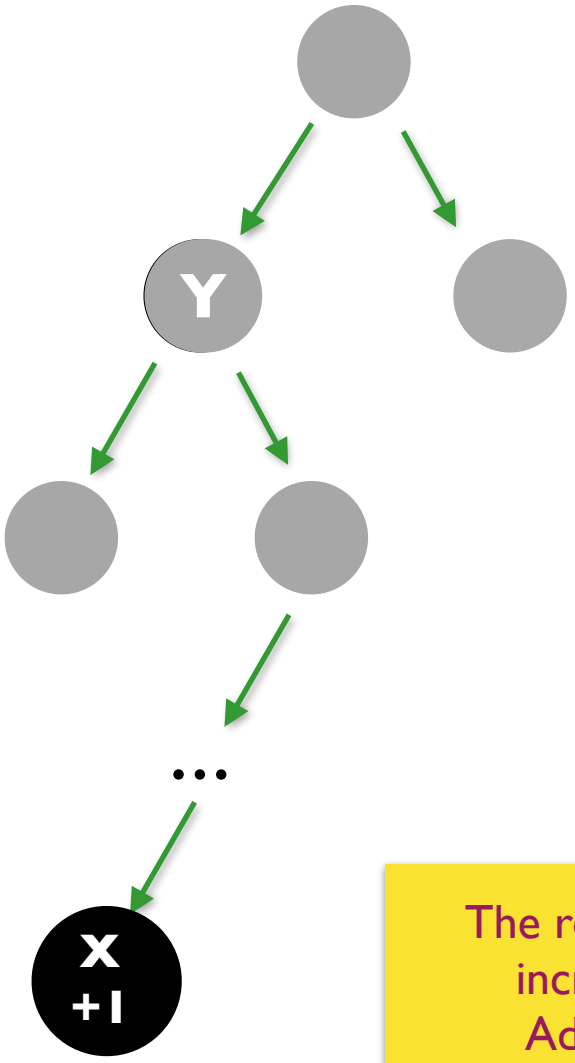
# Deletion



# Deletion



# Deletion




The replacing node X can have increased "black weights". Adjustment is necessary.

- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  1. Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  2. Delete the replacing node from the subtree.

Keep the color

- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

Must *add 1 black* to the path of the replacing node.

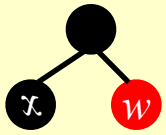
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

Must *add 1 black* to the path of the replacing node.

Case 1:



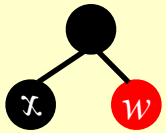
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

Must *add 1 black* to the path of the replacing node.

Case 1:




**Cases are defined by sibling colors**

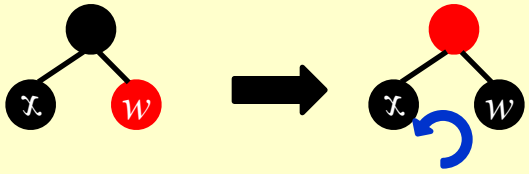
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

Must *add 1 black* to the path of the replacing node.

Case 1:




**Cases are defined by sibling colors**

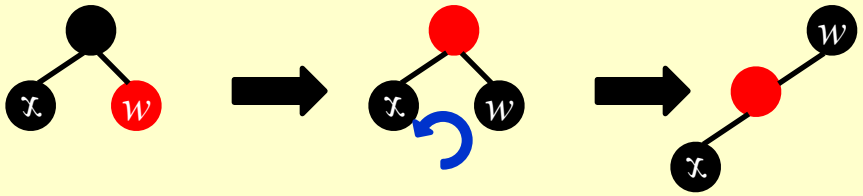
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

Must *add 1 black* to the path of the replacing node.

Case 1:




**Cases are defined by sibling colors**

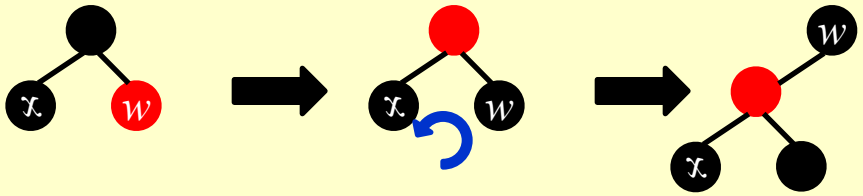
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

Must *add 1 black* to the path of the replacing node.

Case 1:




**Cases are defined by sibling colors**

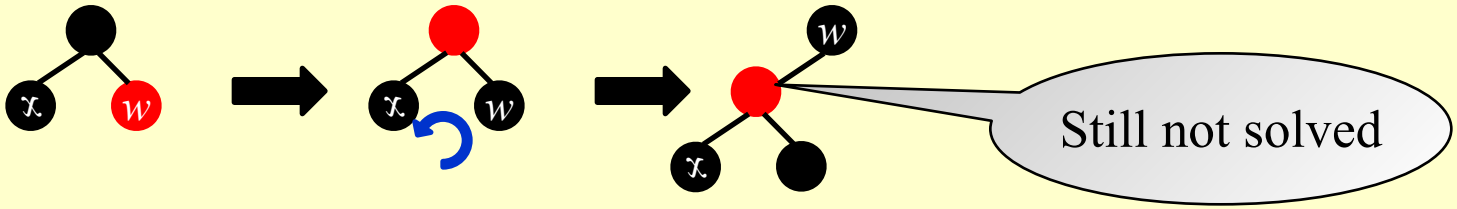
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

Must *add 1 black* to the path of the replacing node.

Case 1:




Still not solved

**Cases are defined by sibling colors**

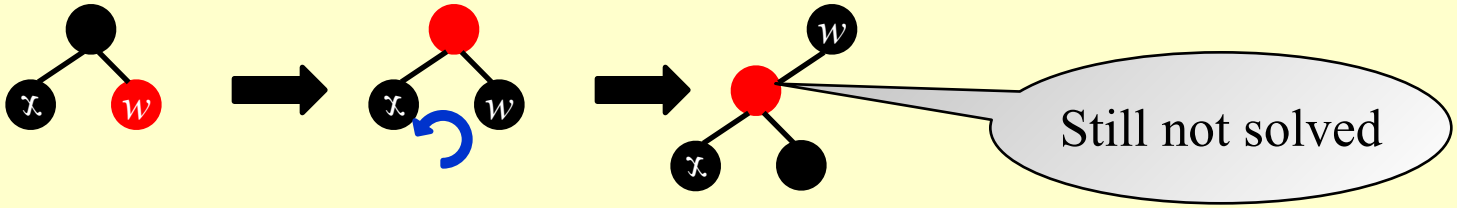
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

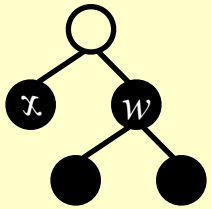
Must *add 1 black* to the path of the replacing node.

Case 1:




Still not solved

Case 2:




**Cases are defined by sibling colors**

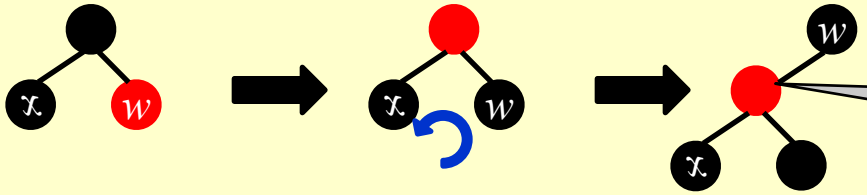
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

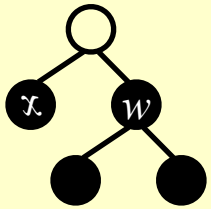
Must *add 1 black* to the path of the replacing node.

Case 1:

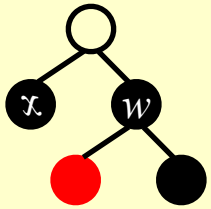



Still not solved

Case 2:




Case 3:




**Cases are defined by sibling colors**

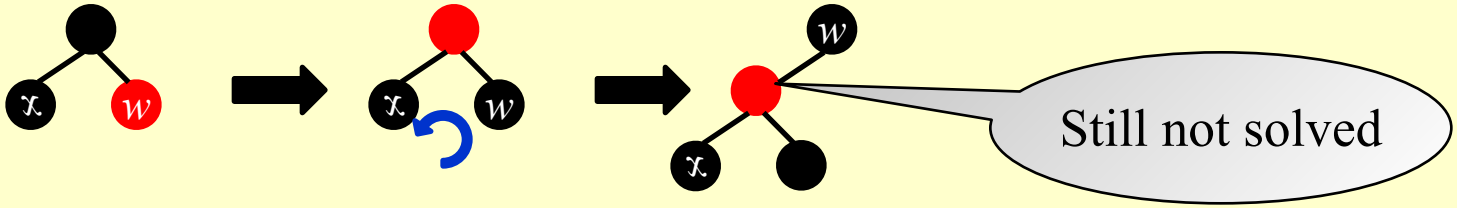
- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :
  ① Replace the node by the largest one in its left subtree or the smallest one in its right subtree.
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

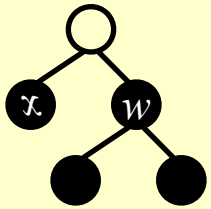
Must *add 1 black* to the path of the replacing node.

Case 1:

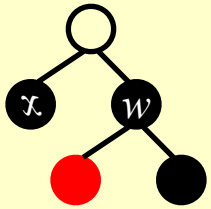



Still not solved

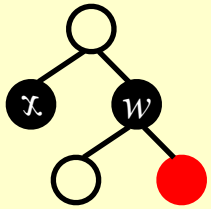
Case 2:




Case 3:

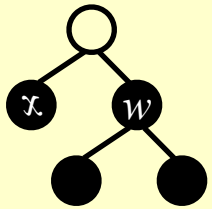



Case 4:

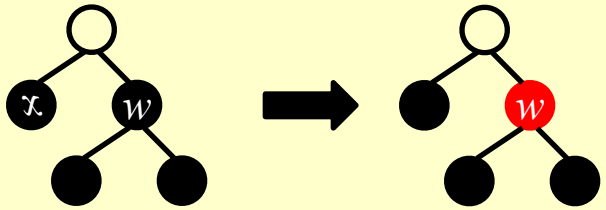



**Cases are defined by sibling colors**

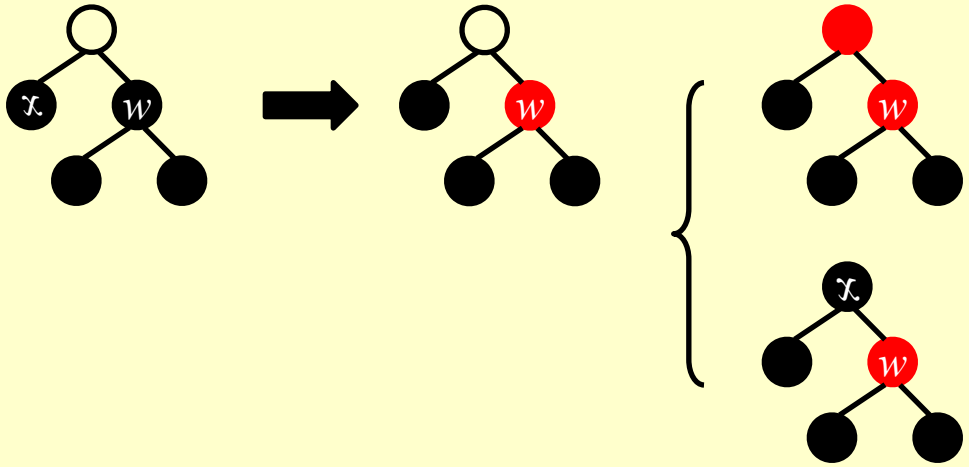
Case 2:



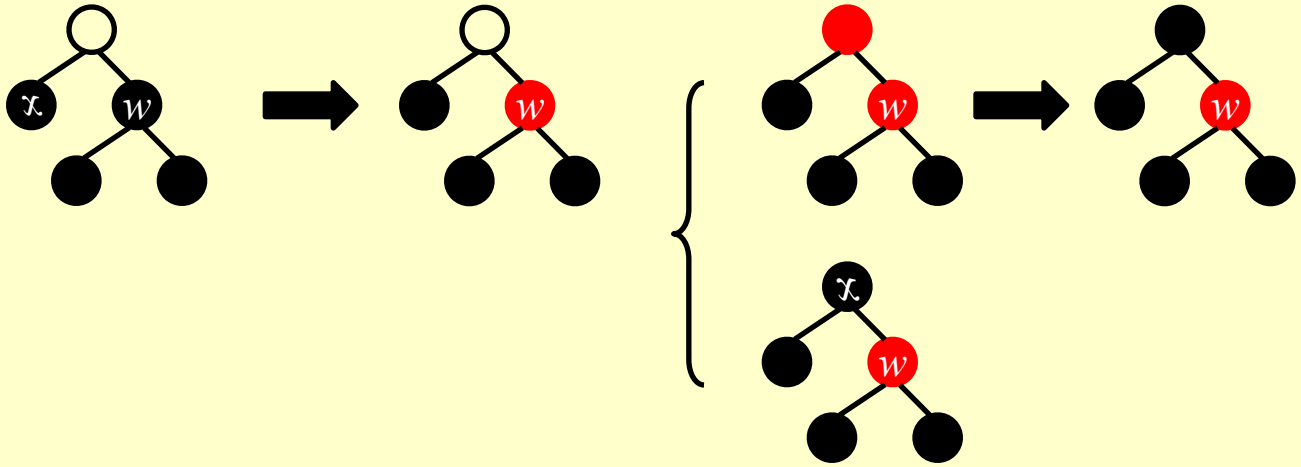
Case 2:



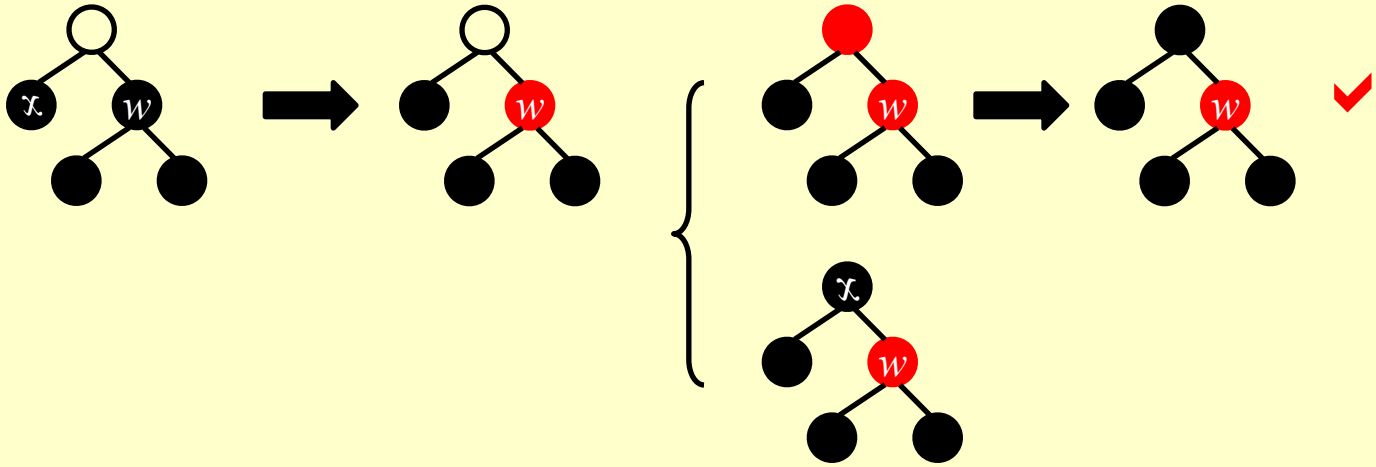
Case 2:



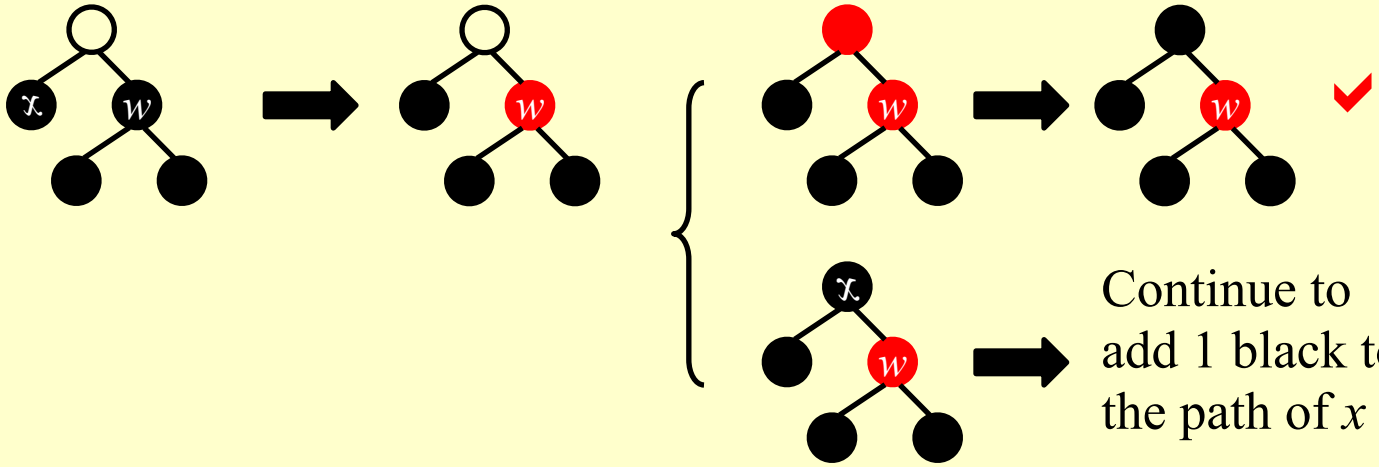
Case 2:



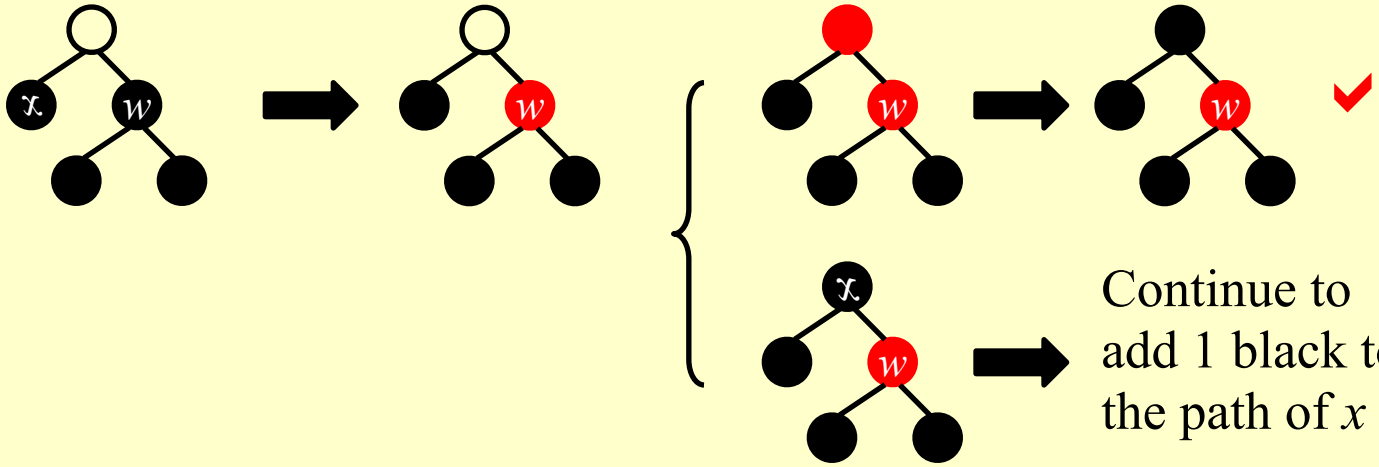
Case 2:



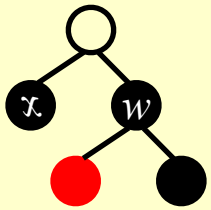
Case 2:



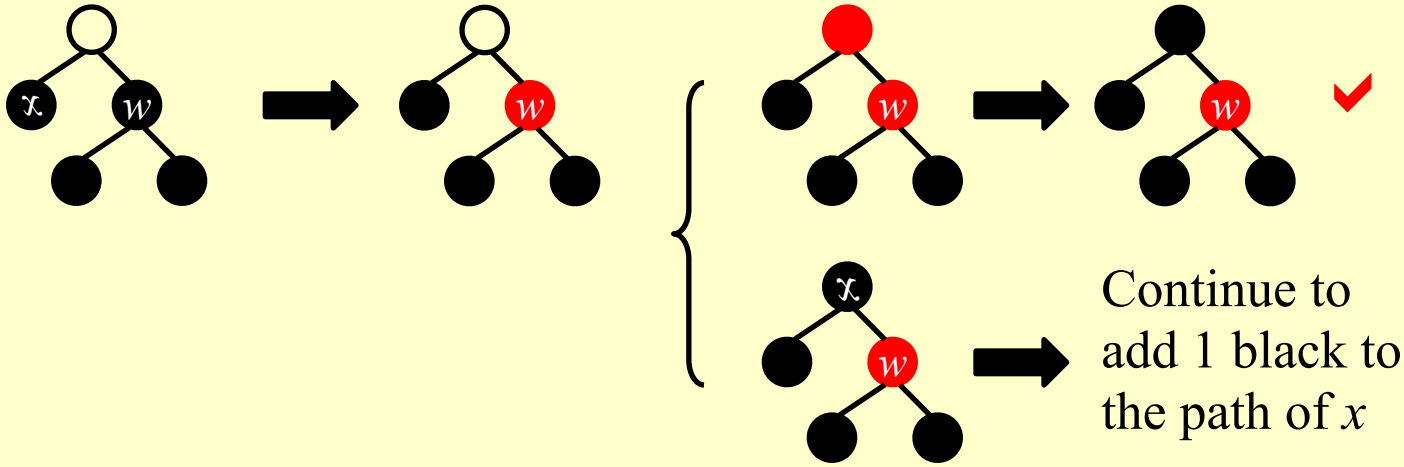
Case 2:



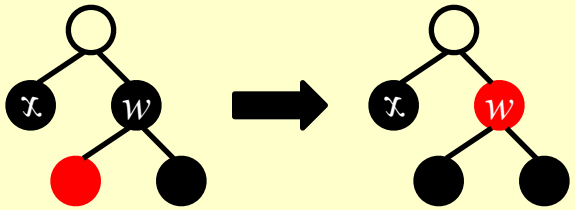

Case 3:



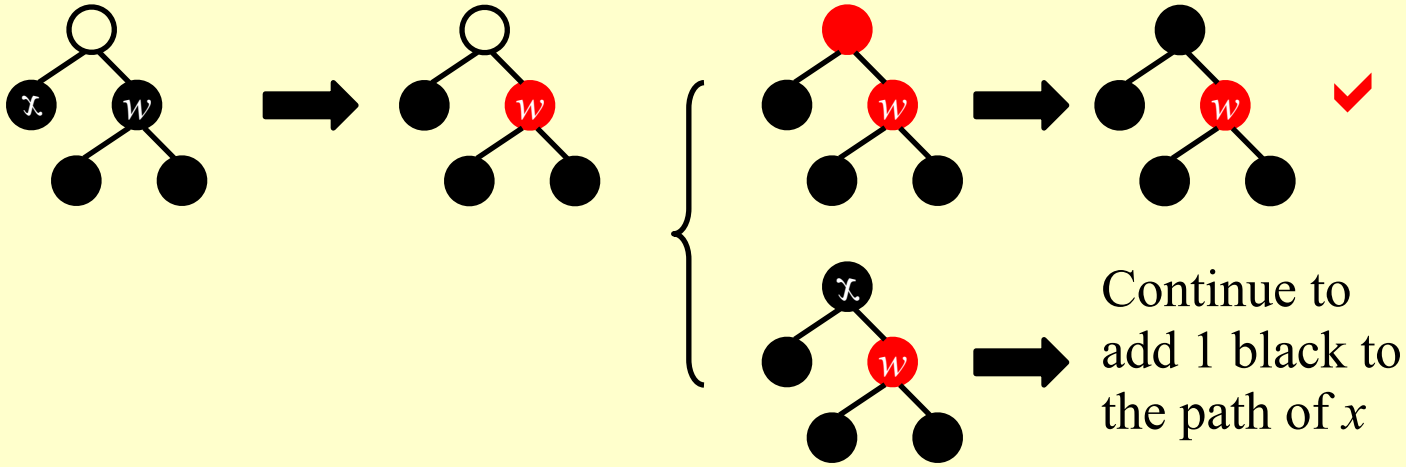
Case 2:



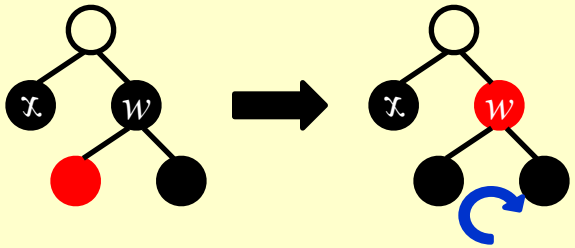

Case 3:



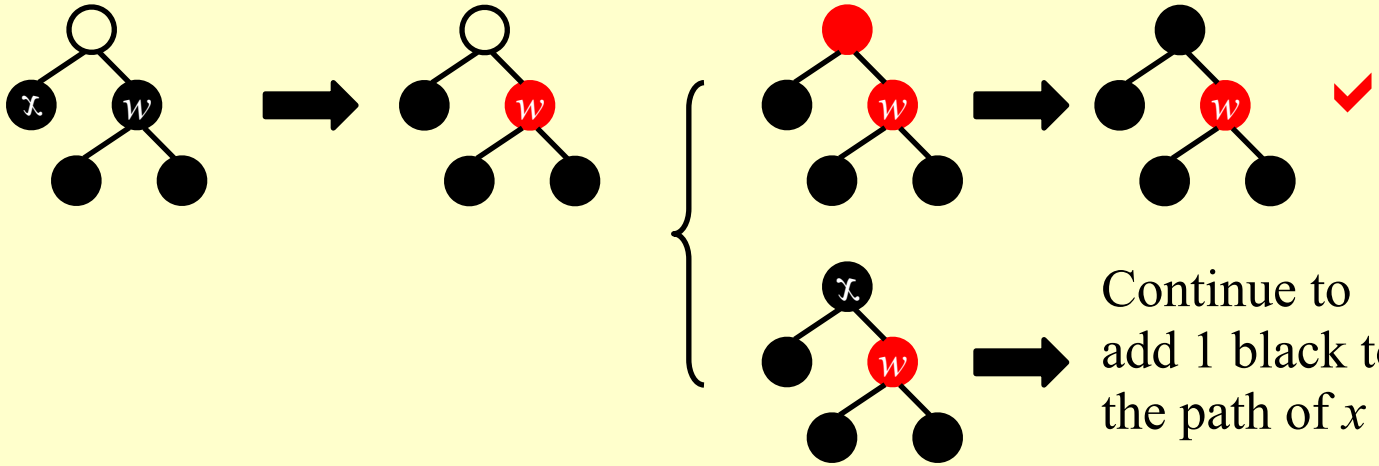
Case 2:




Case 3:

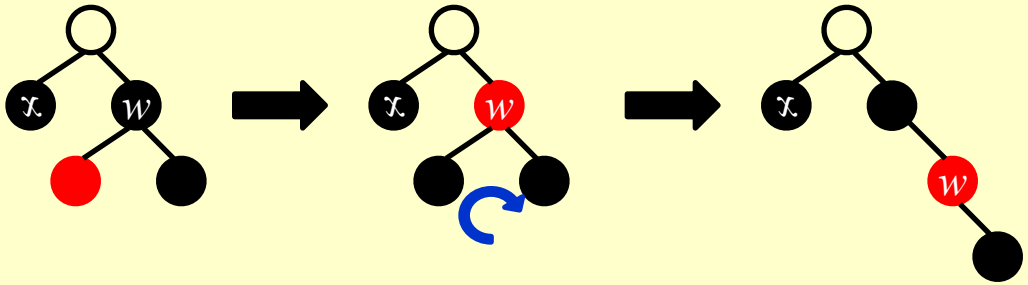


Case 2:

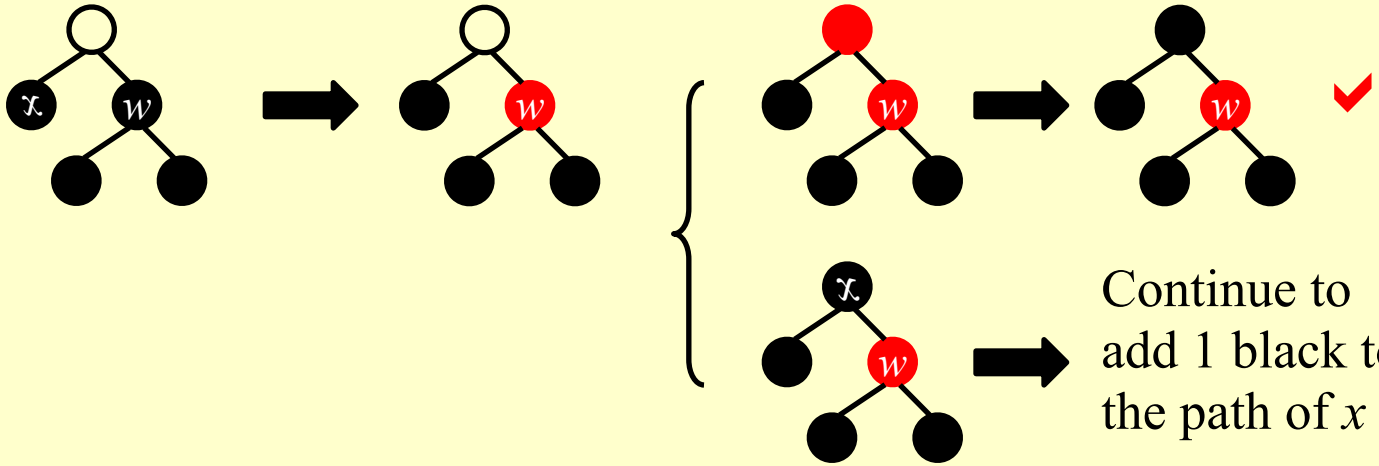



Continue to add 1 black to the path of $x$

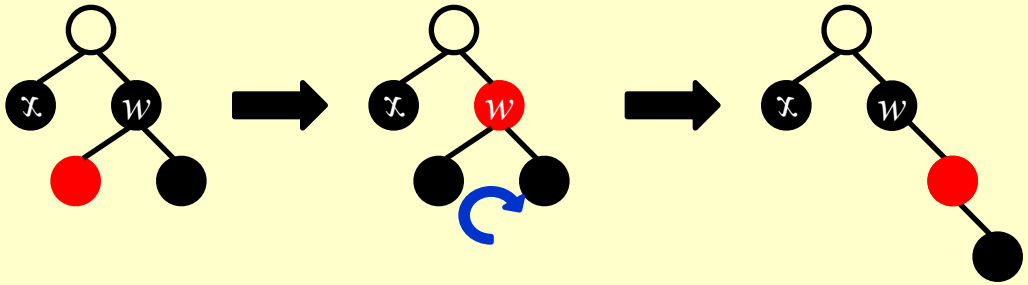
Case 3:



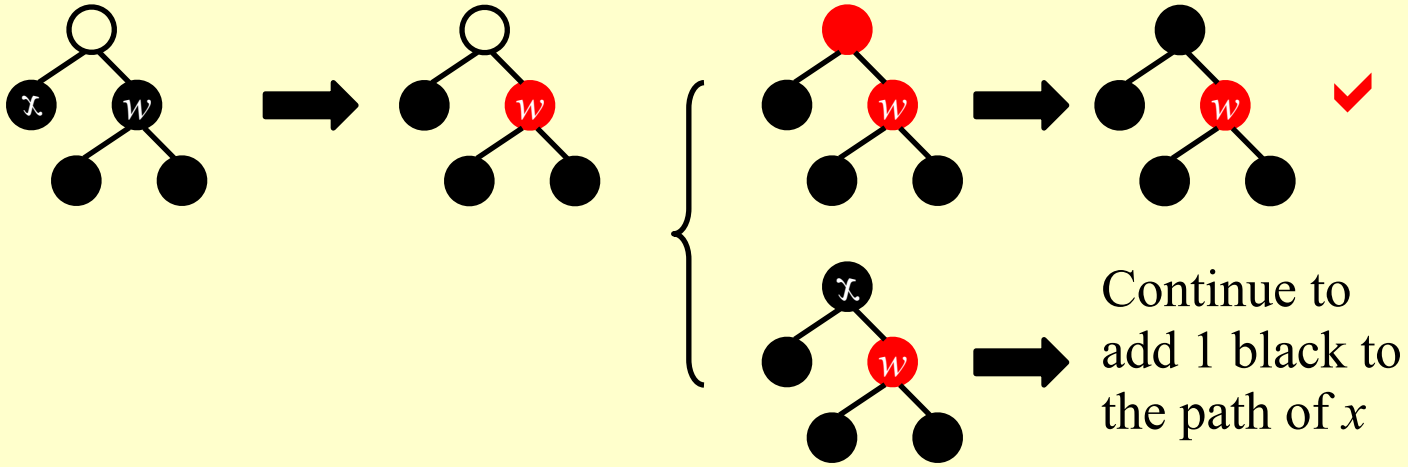
Case 2:




Case 3:

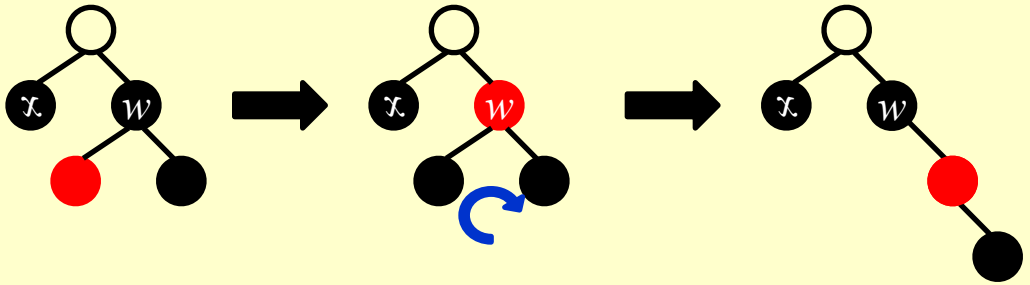


Case 2:



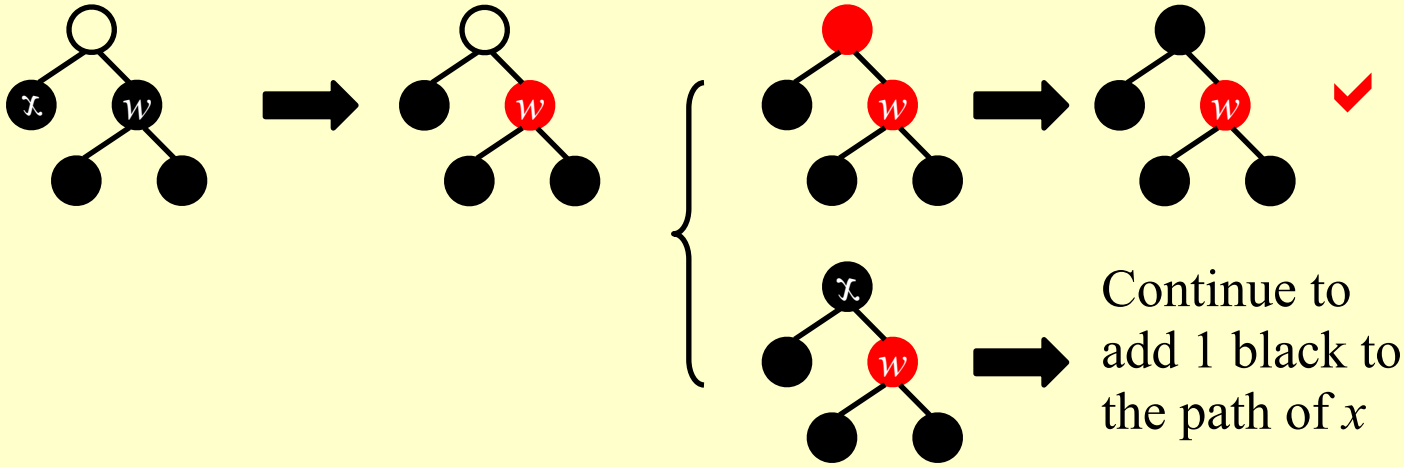

Continue to add 1 black to the path of $x$

Case 3:



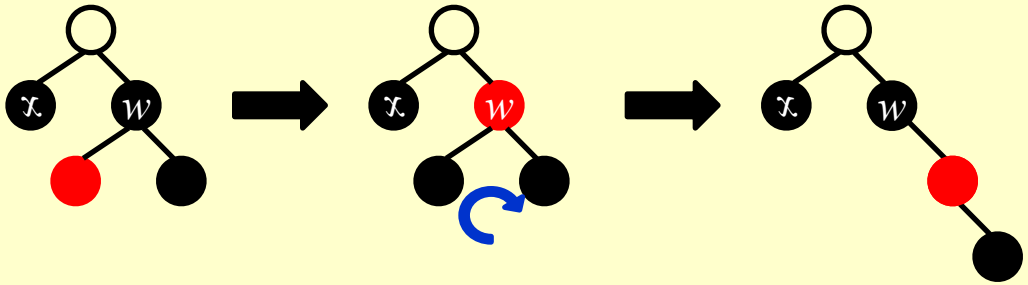

Case 4

Case 2:



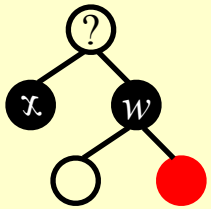

Continue to add 1 black to the path of $x$

Case 3:

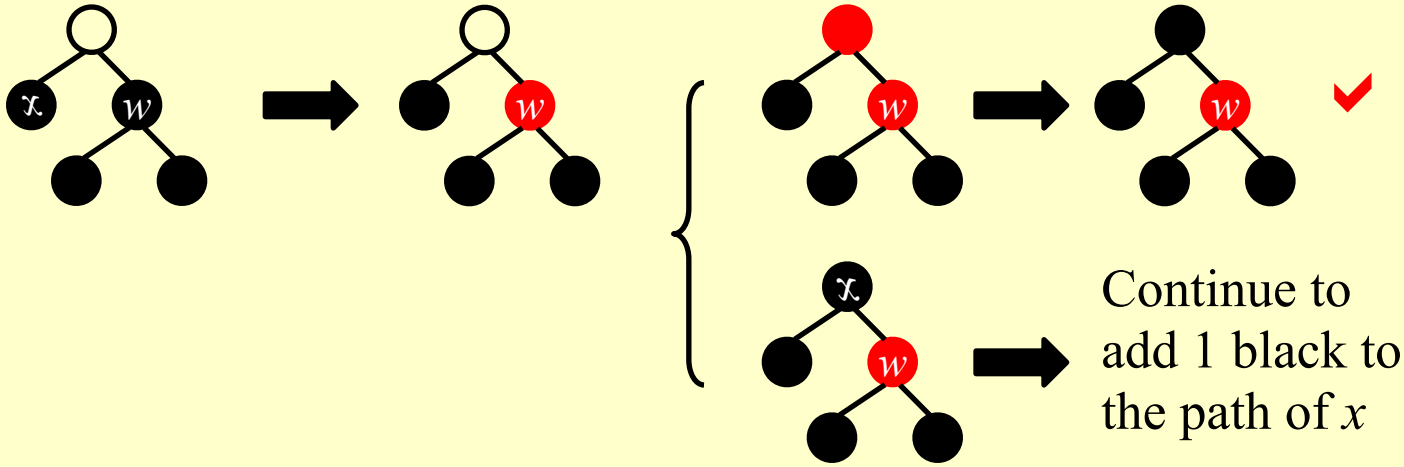



Case 4

Case 4:

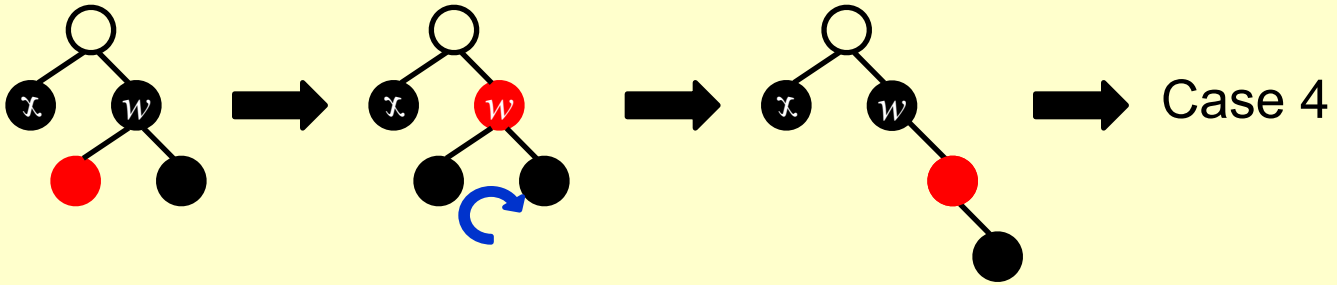


Case 2:



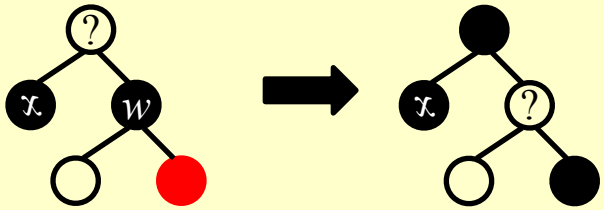

Continue to add 1 black to the path of $x$

Case 3:

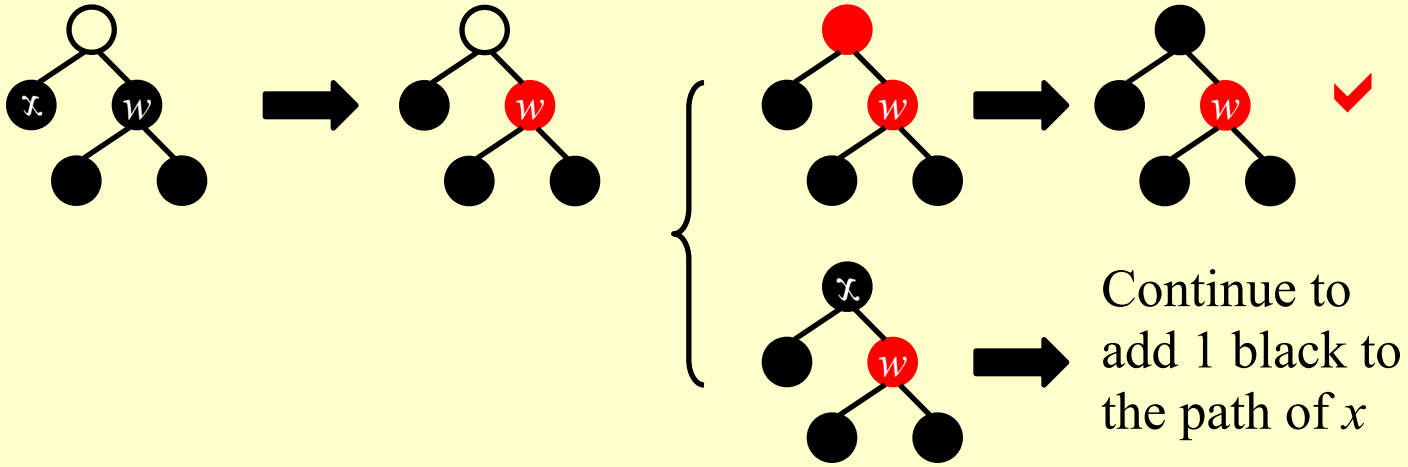



Case 4

Case 4:

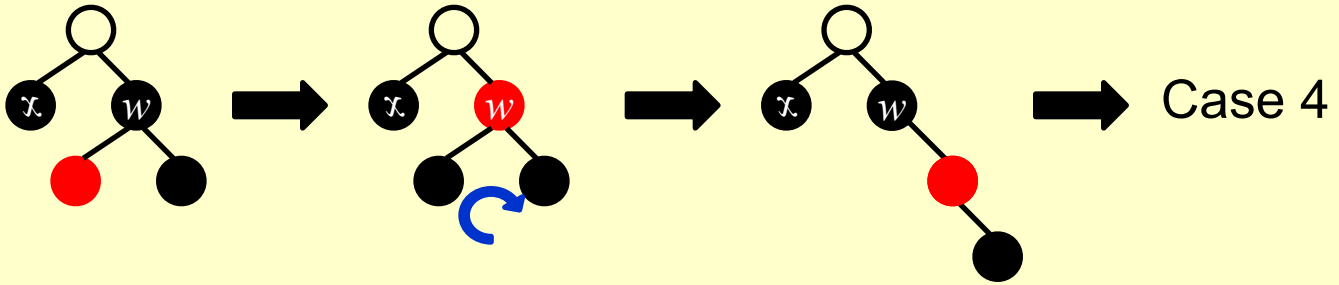


Case 2:



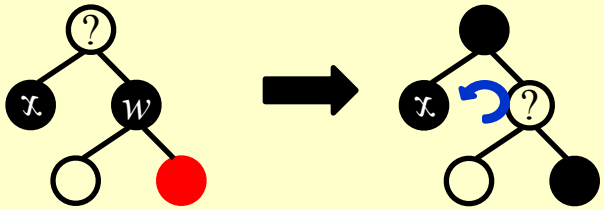

Continue to add 1 black to the path of $x$

Case 3:

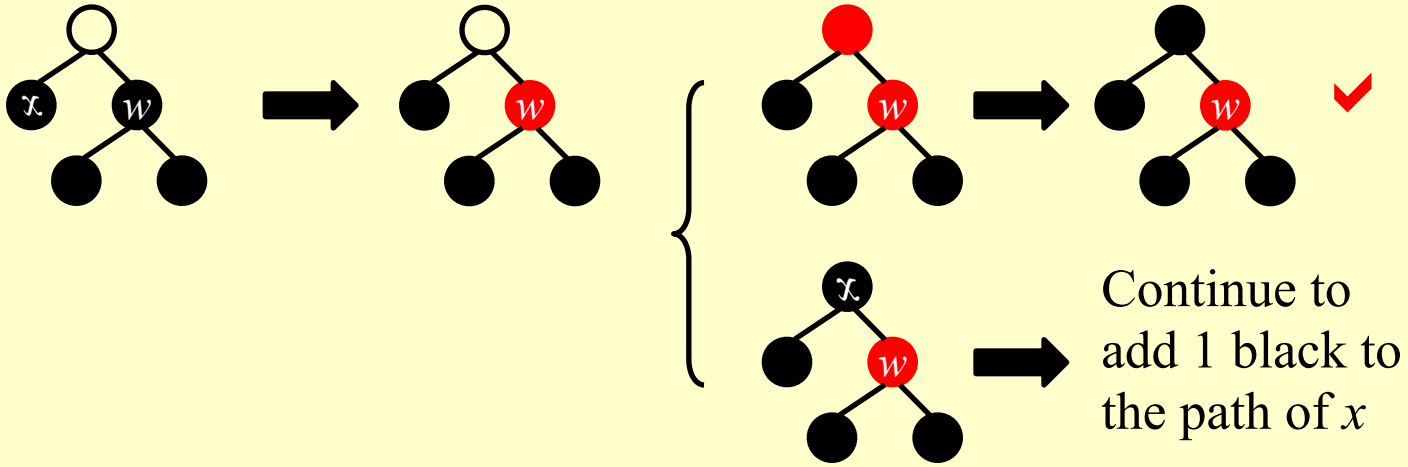



Case 4

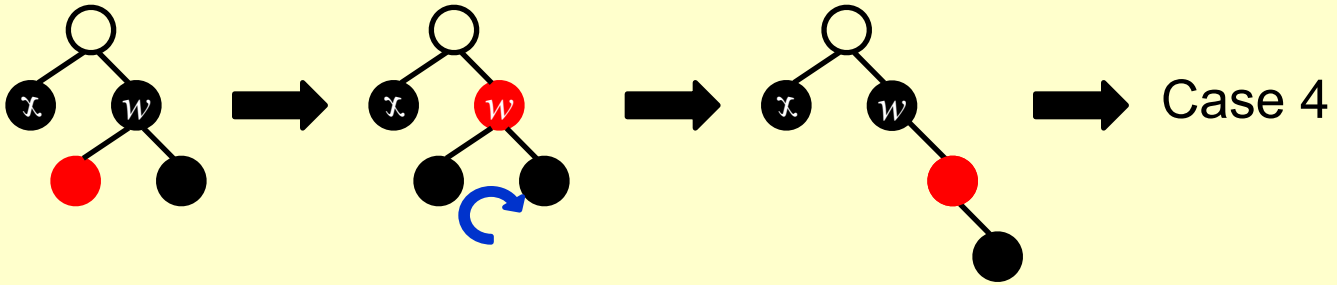
Case 4:



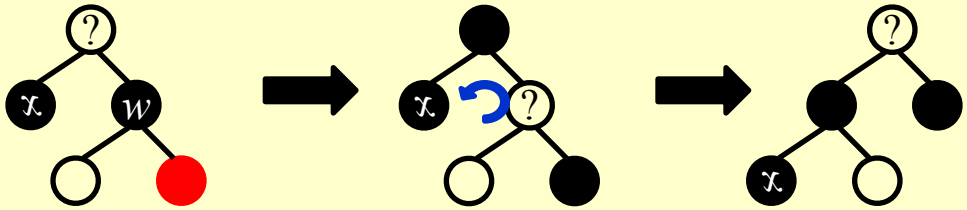
Case 2:



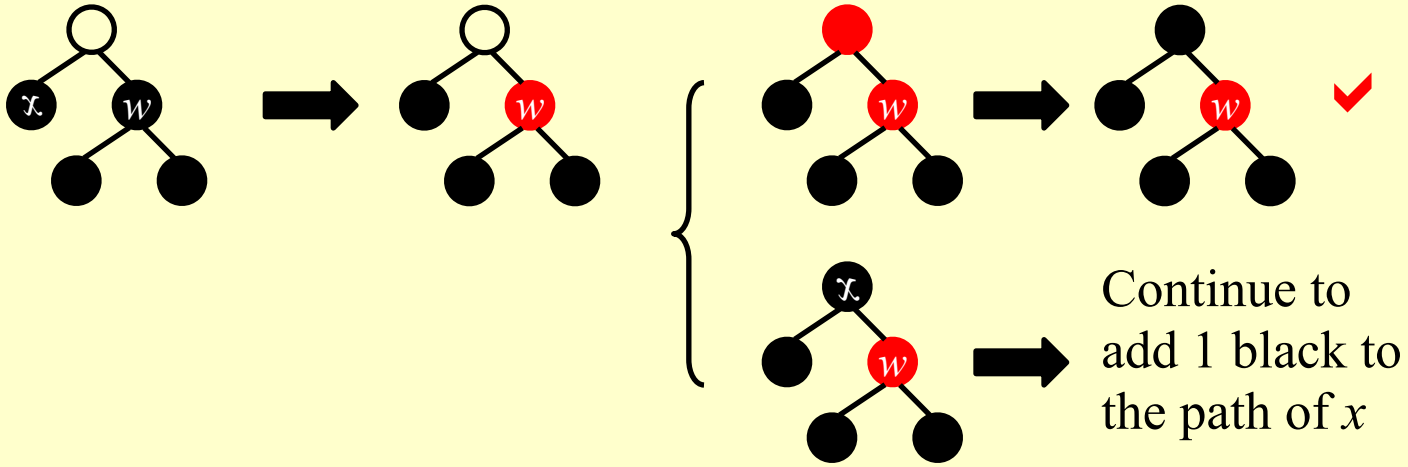

Case 3:




Case 4:

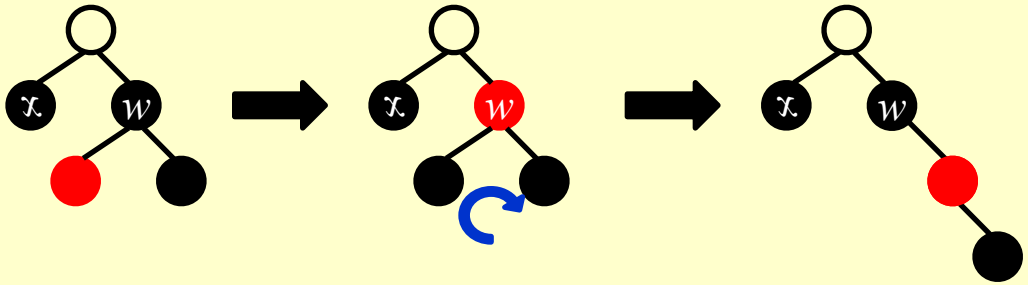


Case 2:



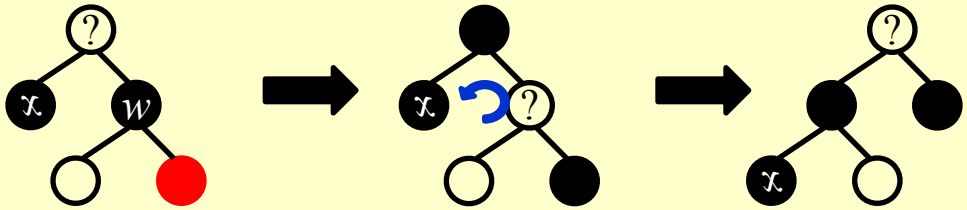

Continue to add 1 black to the path of $x$

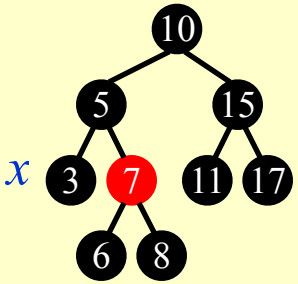
Case 3:



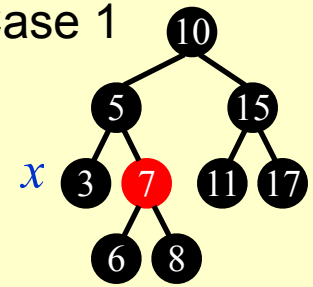

Case 4

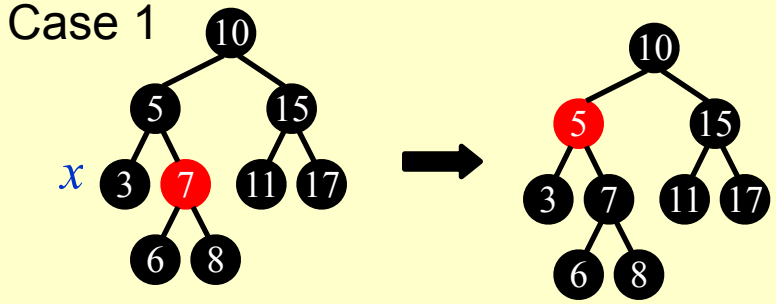
Case 4:

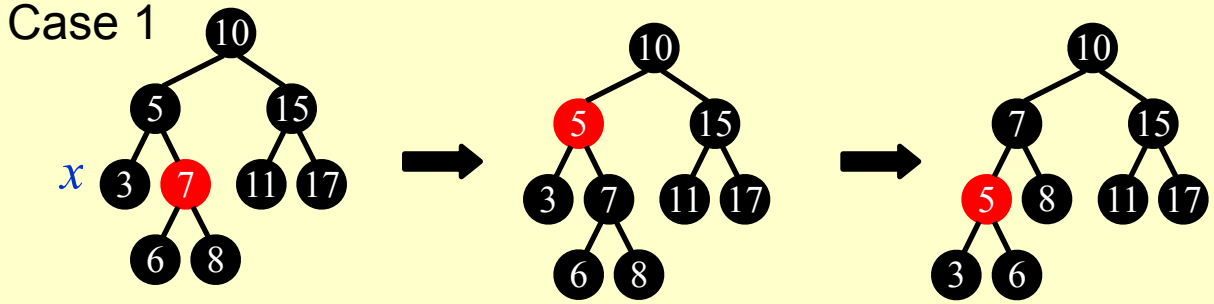


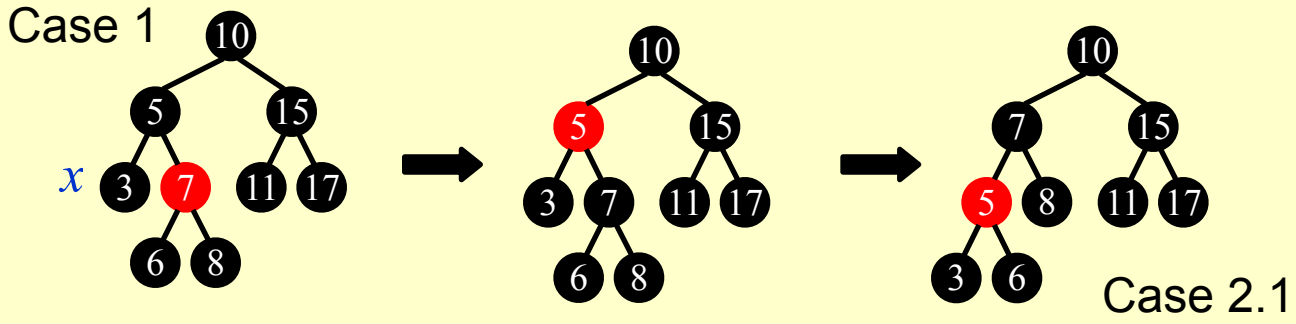

10
5
15
x
3
7
11
17
6
8

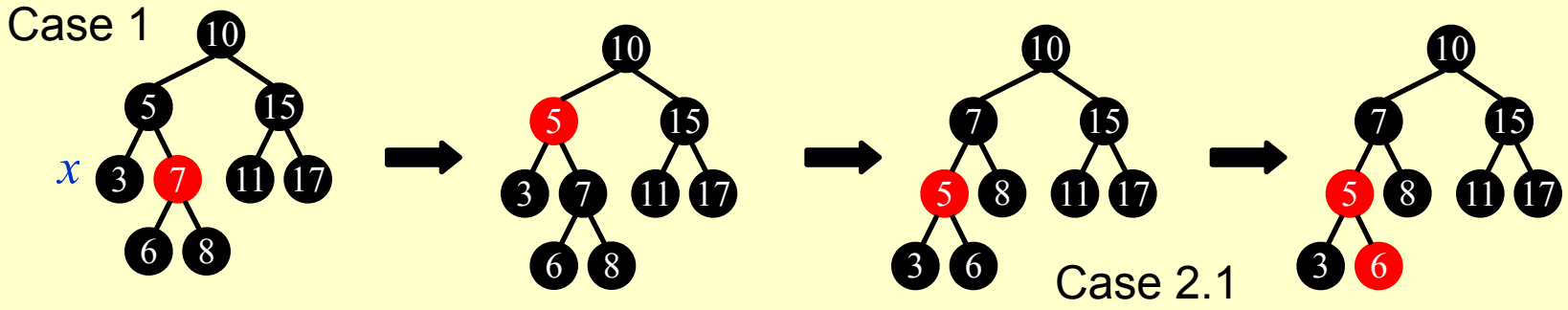
Case 1

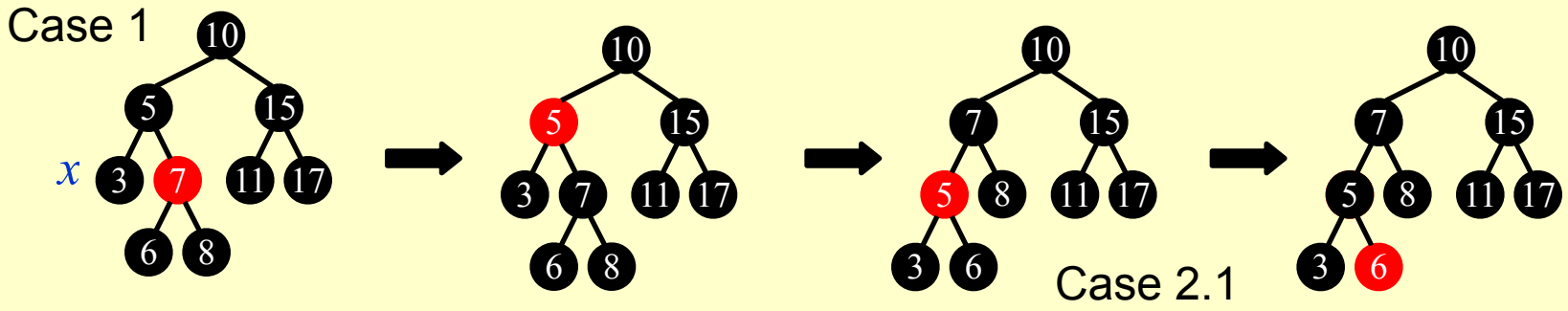


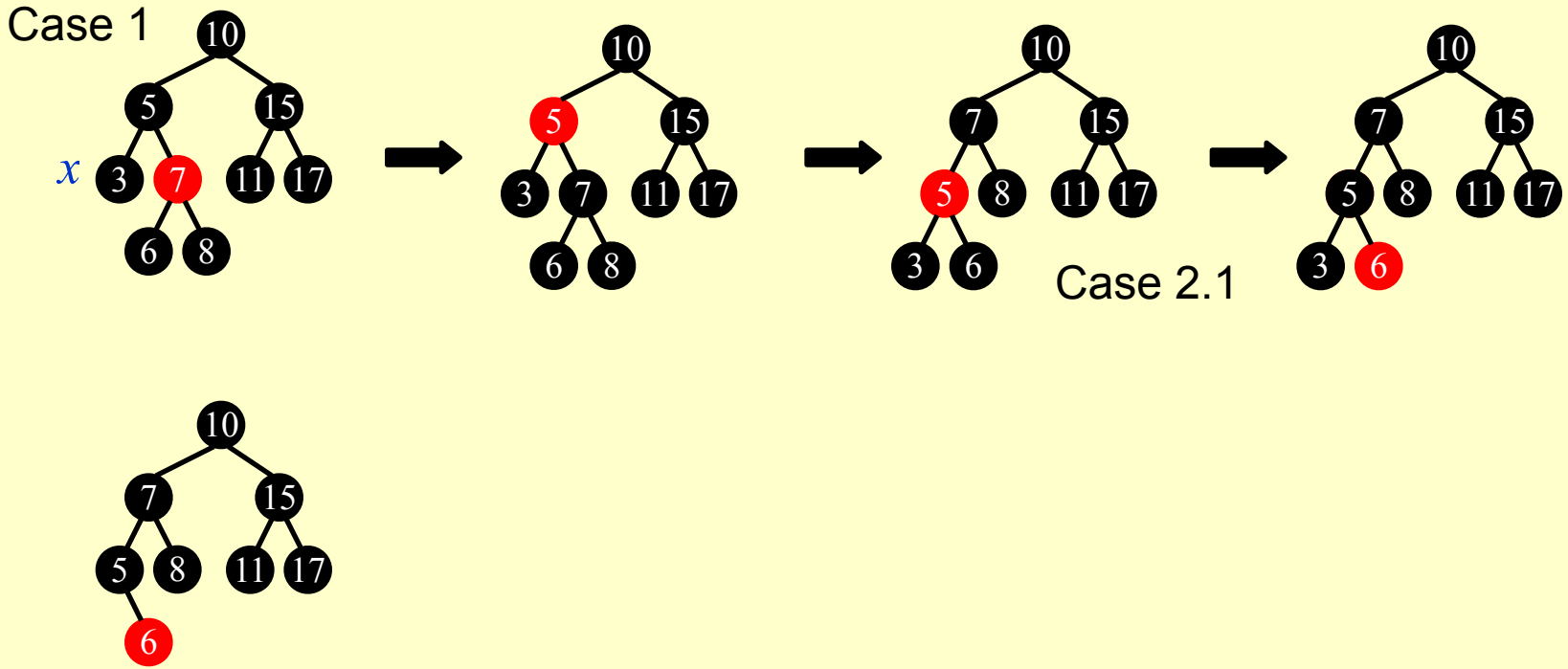

Case 1
10
5
15
x
3
7
11
17
6
8
10
5
15
3
7
11
17
6
8

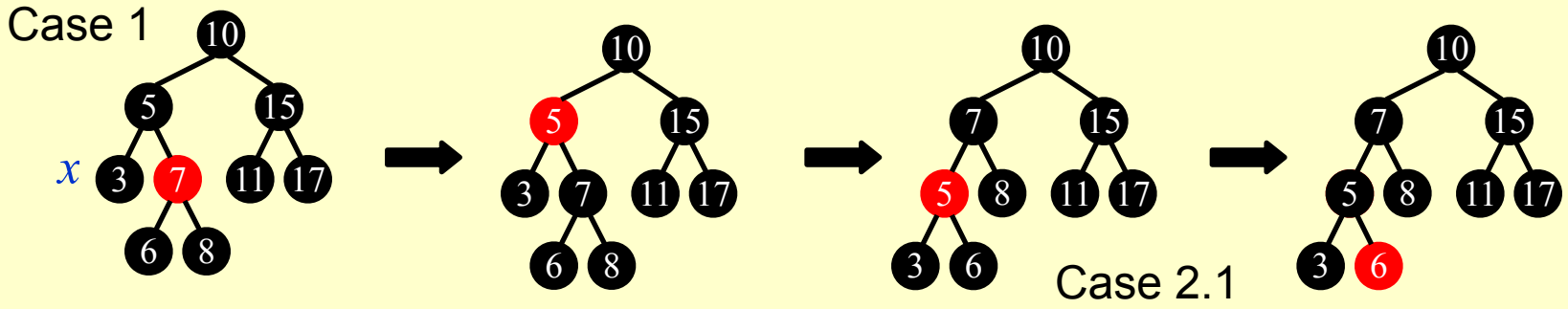

Case 1
x
10
5
15
3
7
11
17
6
8
10
5
15
3
7
11
17
6
8
10
7
15
5
8
11
17
3
6

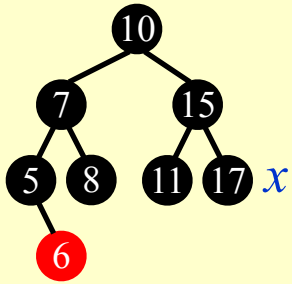

Case 1
10
5
15
x
3
7
11
17
6
8
10
5
15
3
7
11
17
6
8
10
7
15
5
8
11
17
3
6
Case 2.1

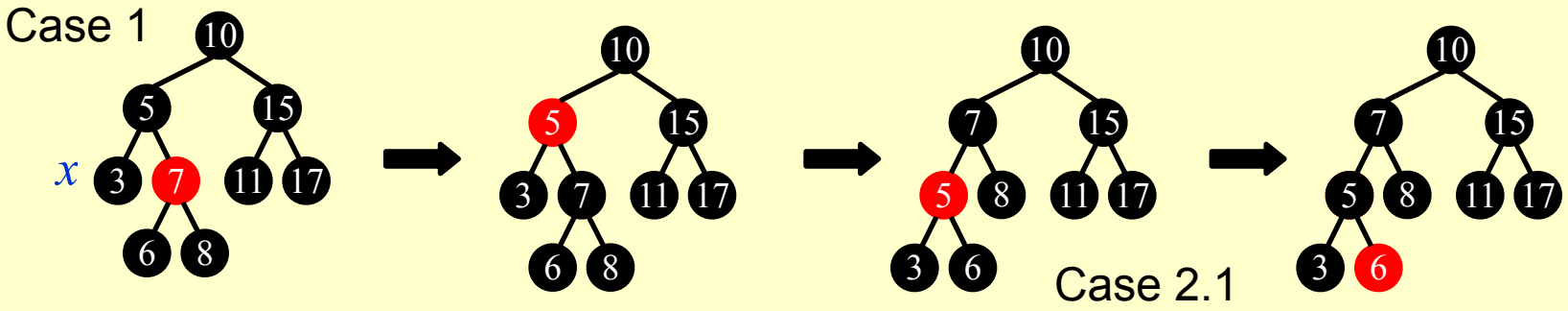

Case 1
x
10
5
15
3
7
11
17
6
8
Case 2.1

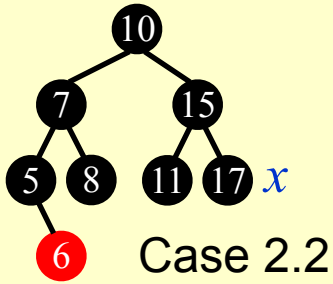

Case 1
x
10
5
15
3
7
11
17
6
8
Case 2.1


Case 1
x
Case 2.1

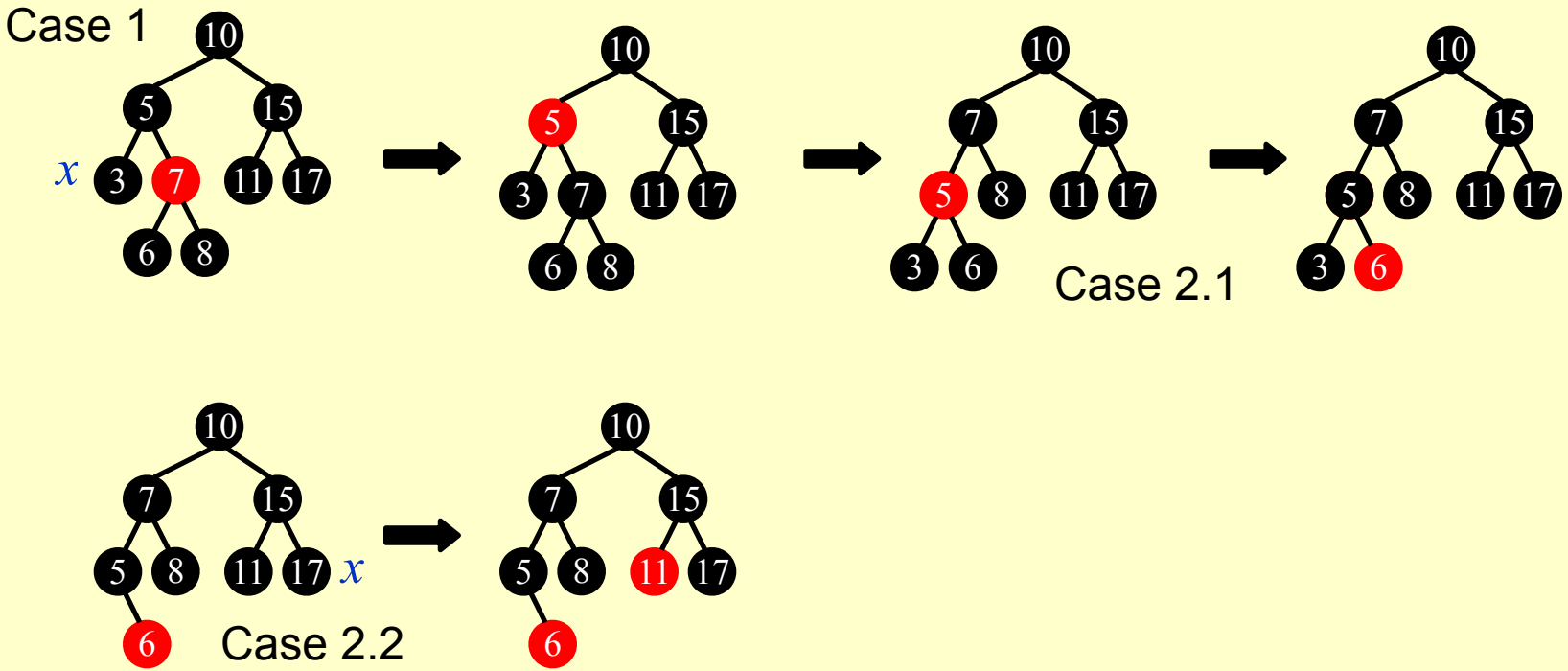

Case 1
10
5
15
x
3
7
11
17
6
8
10
5
15
3
7
11
17
6
8
10
7
15
5
8
11
17
3
6
Case 2.1
10
7
15
5
8
11
17
3
6

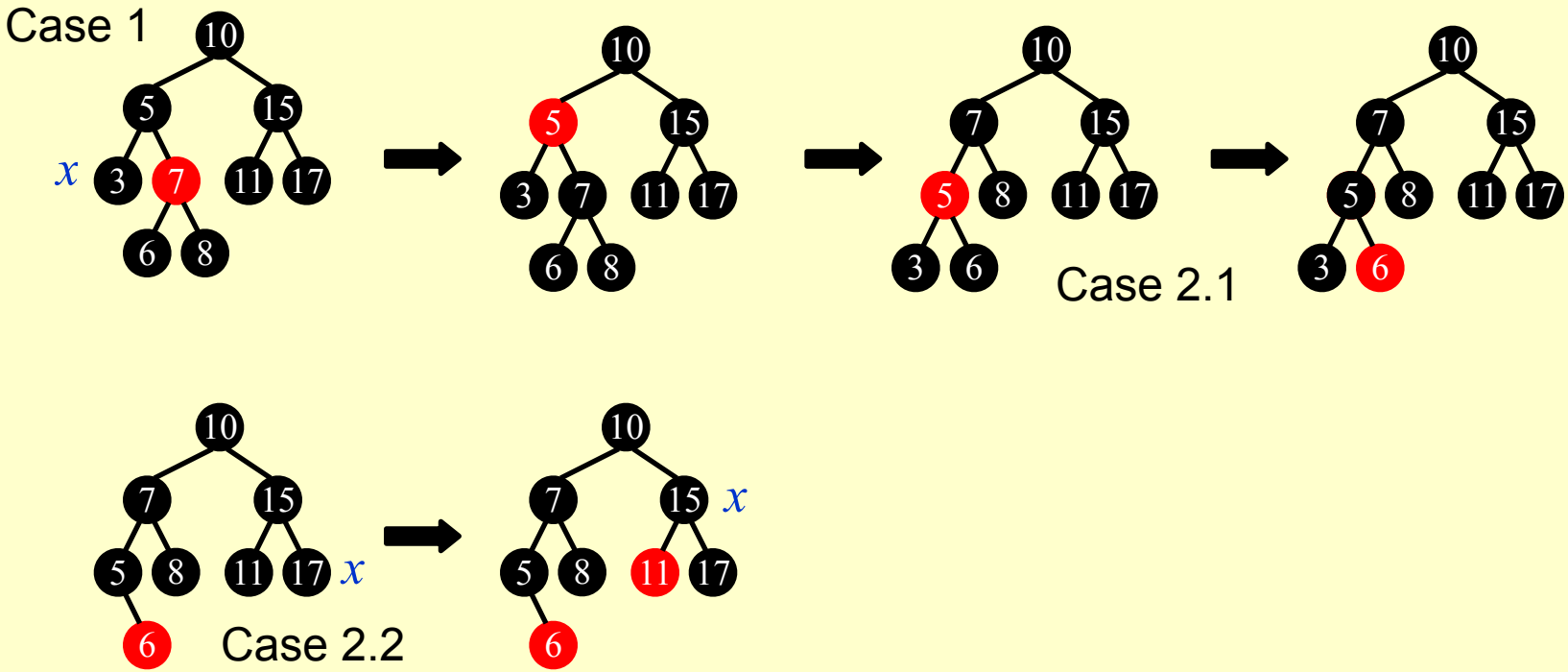

10
7
15
5
8
11
17
x
6

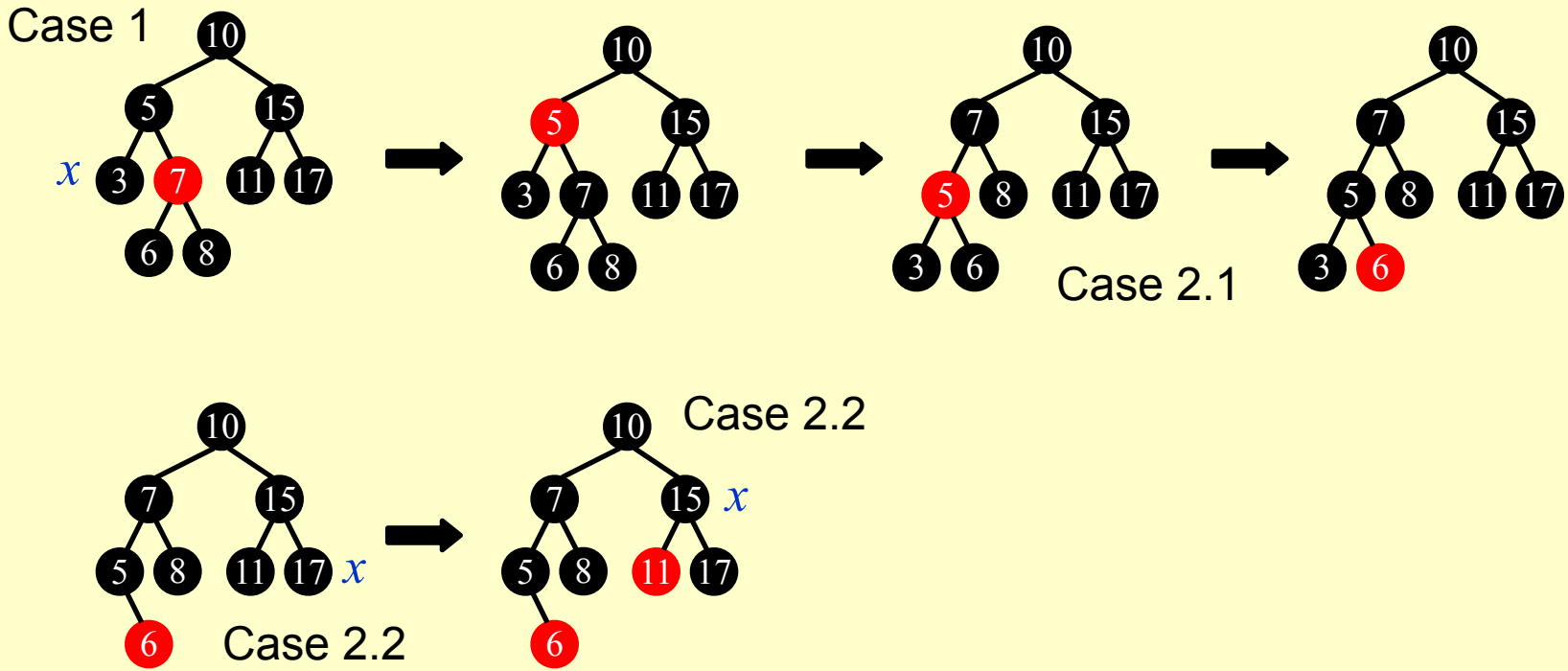


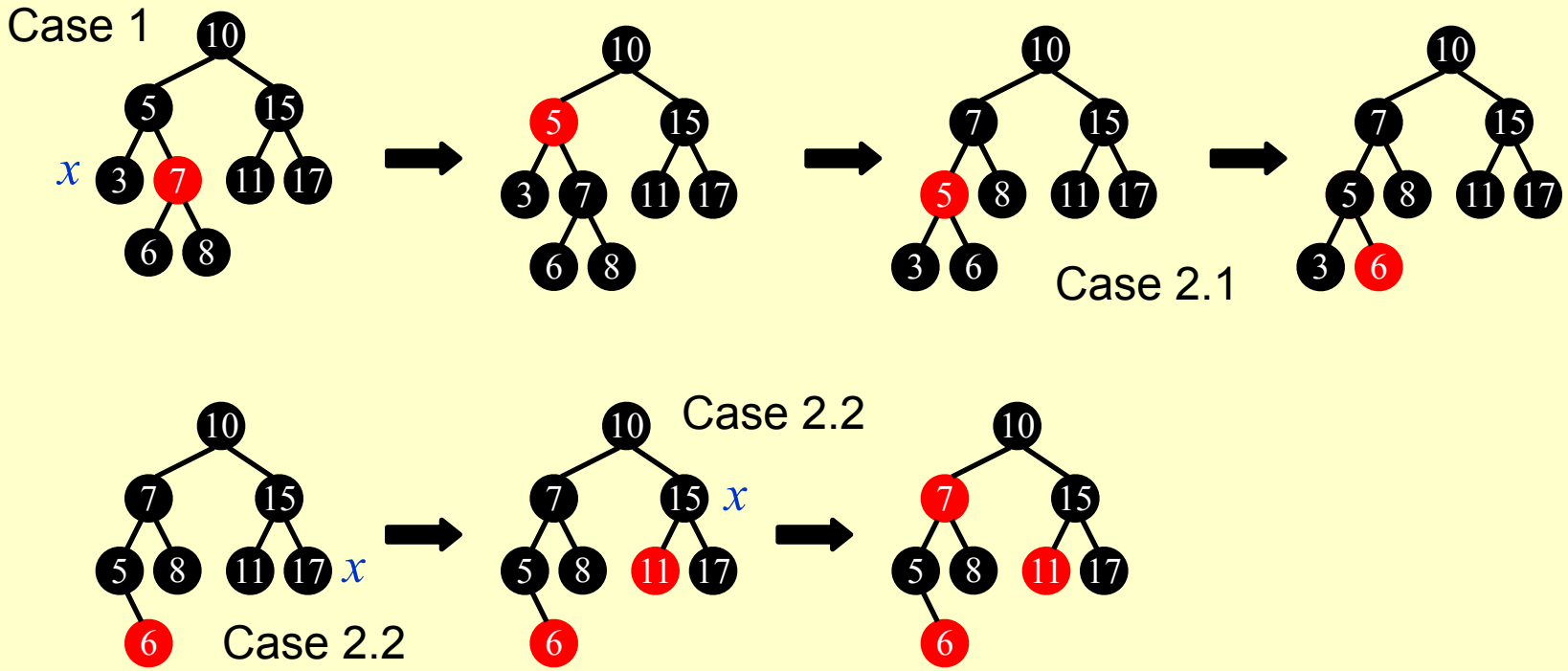



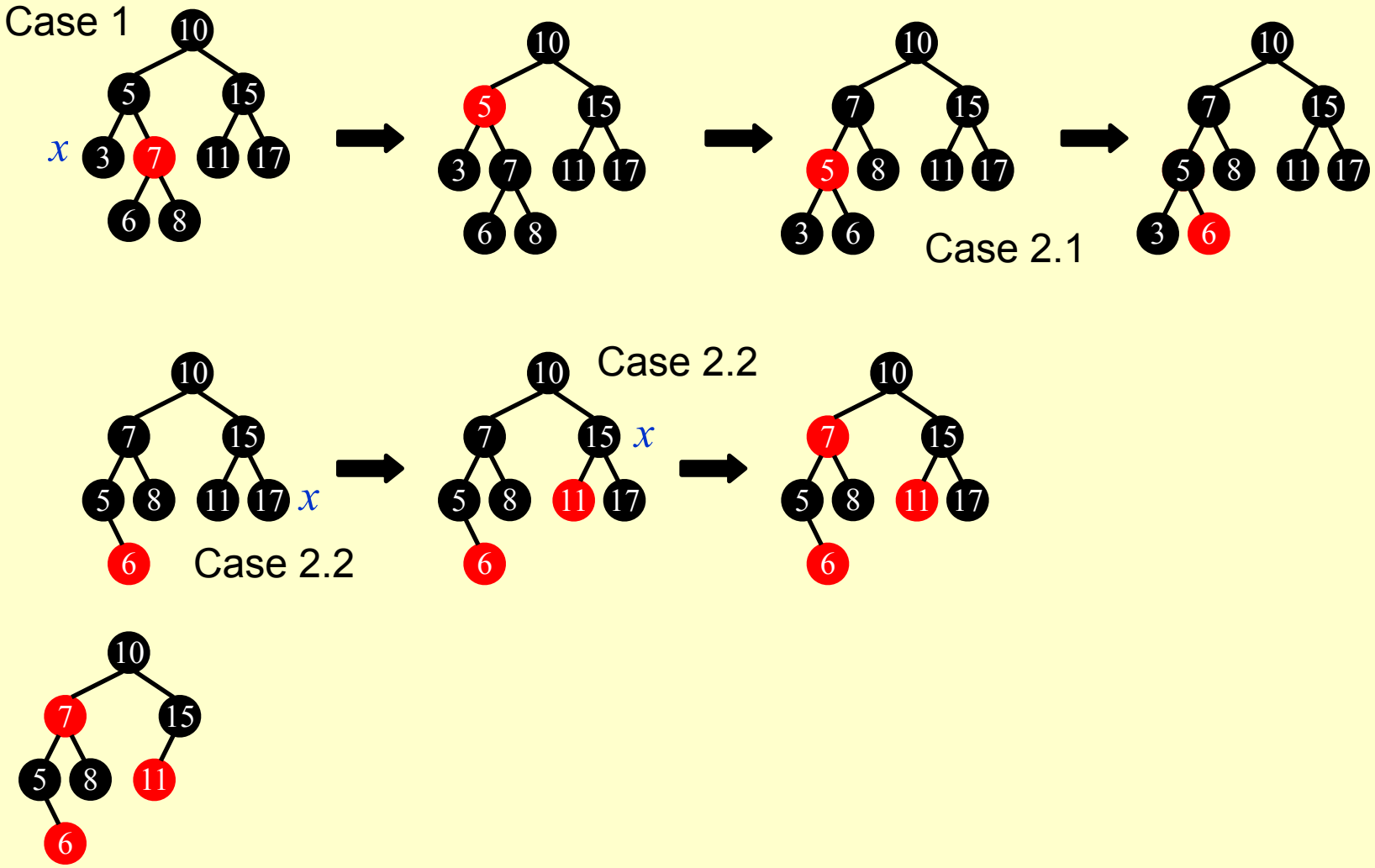

Case 2.2

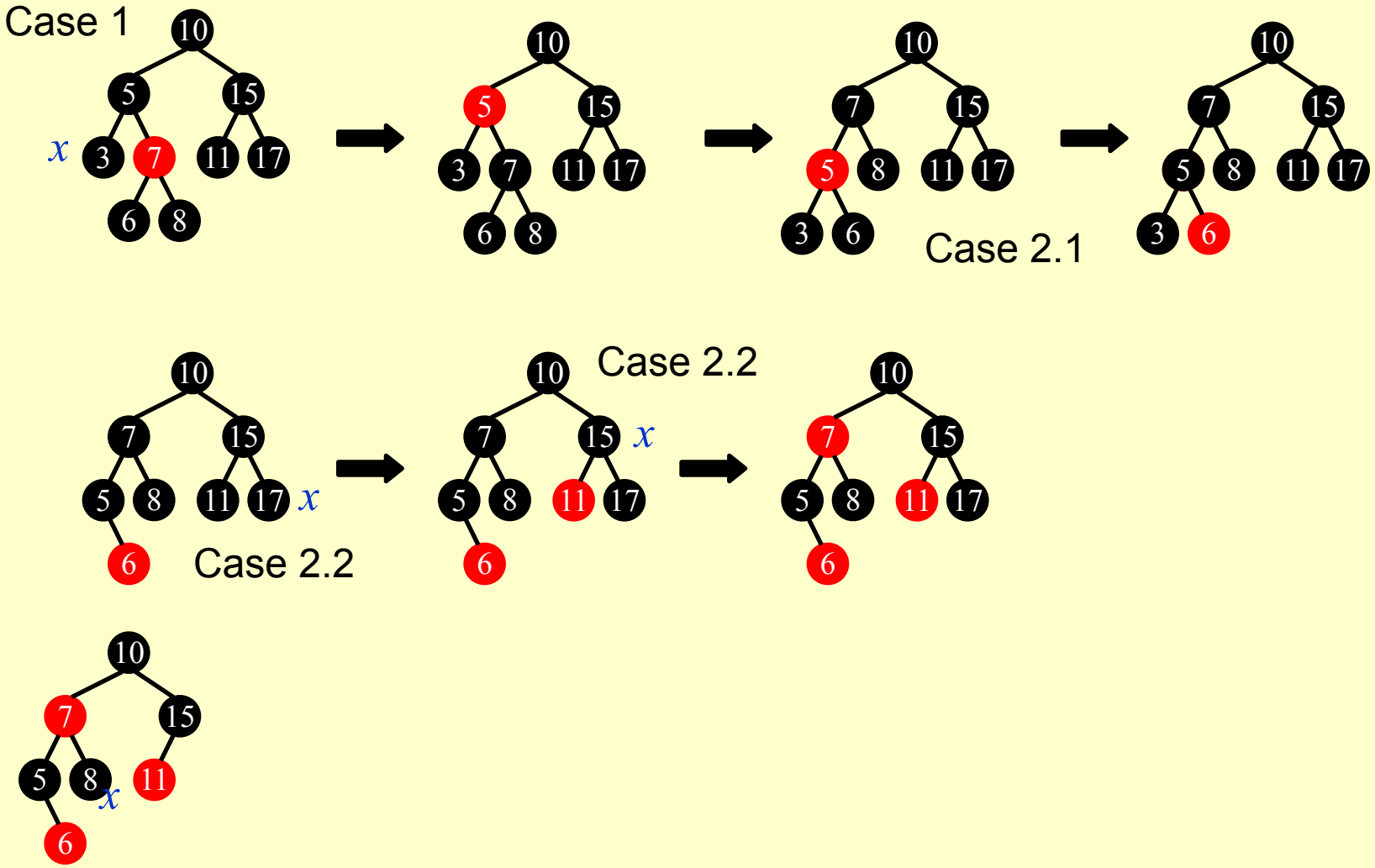

Case 1
x
Case 2.1
Case 2.2
x

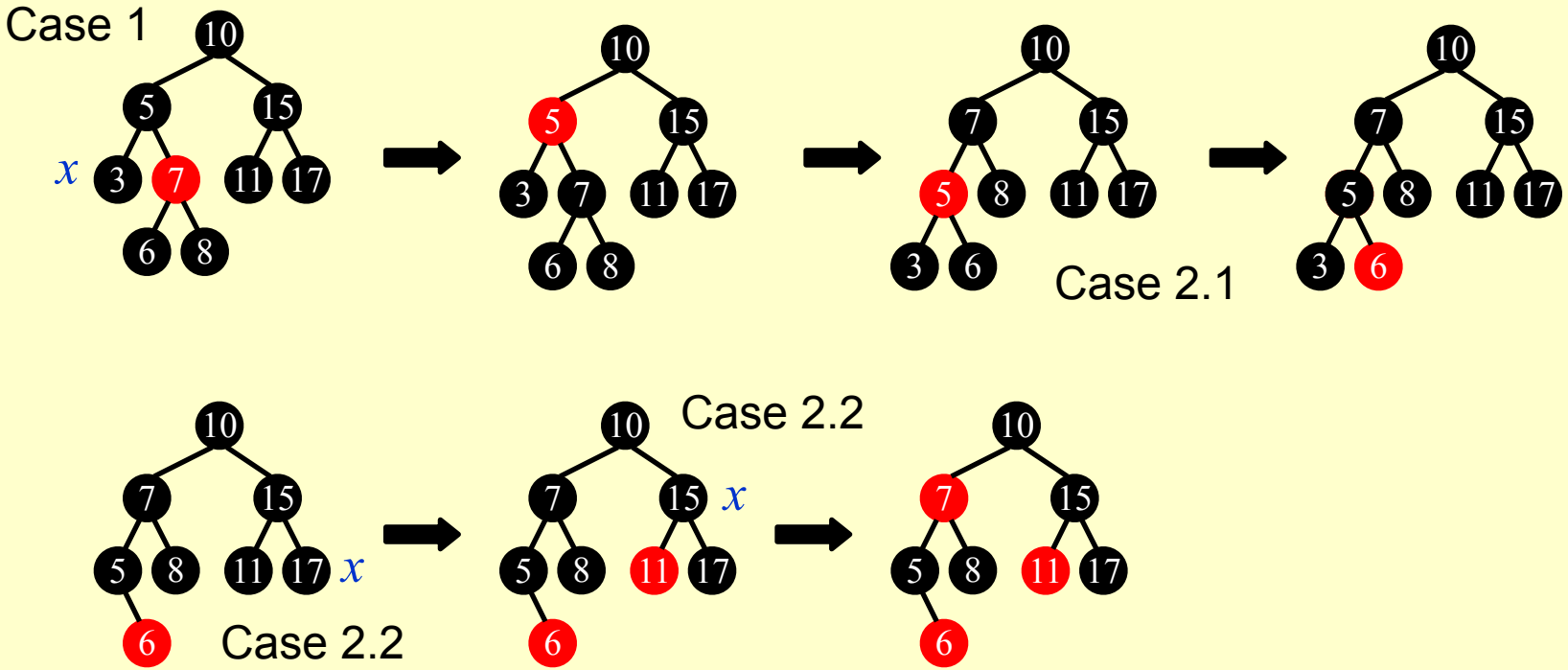

Case 1
x
Case 2.1
Case 2.2
x
x

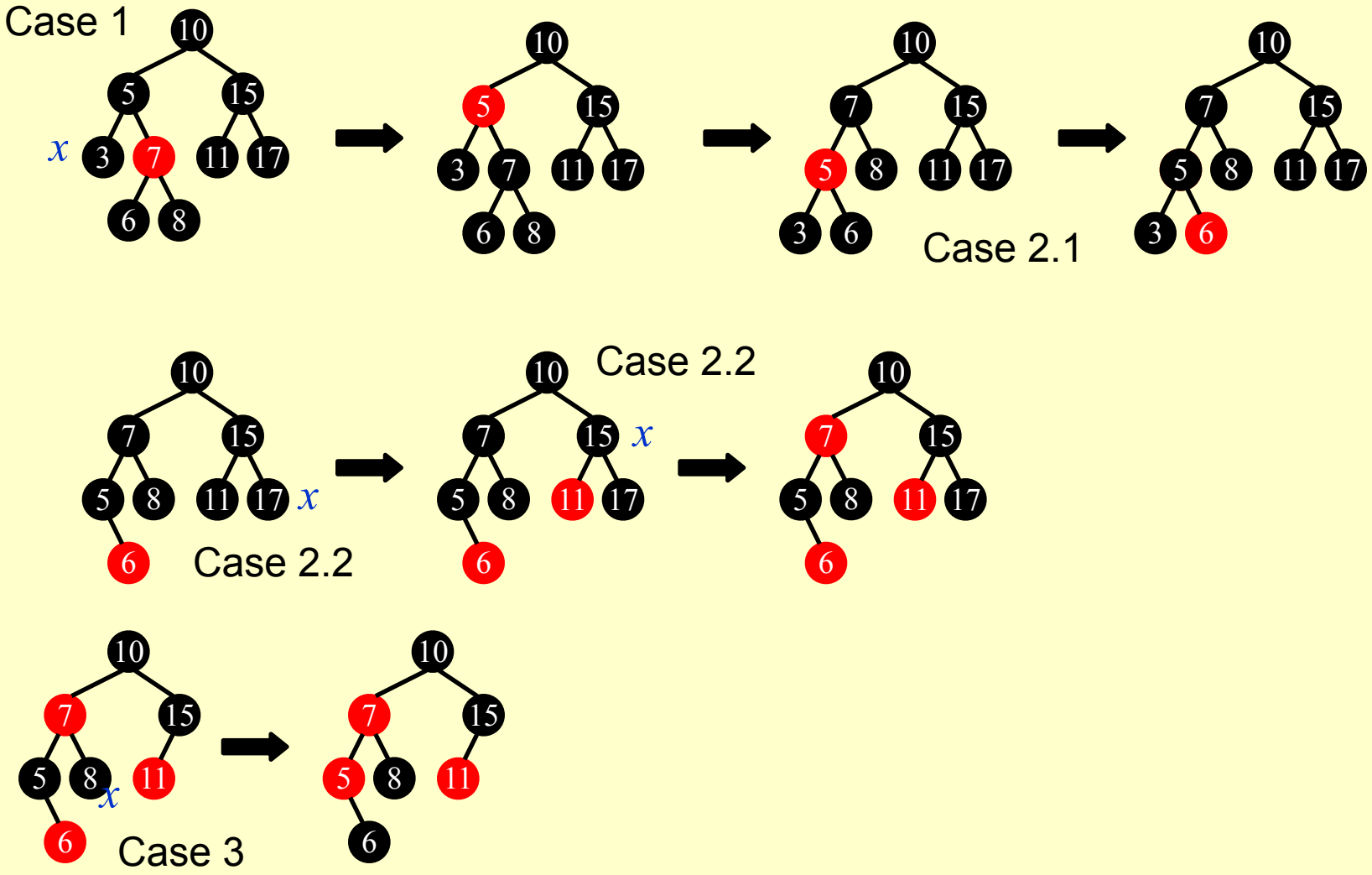

Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x

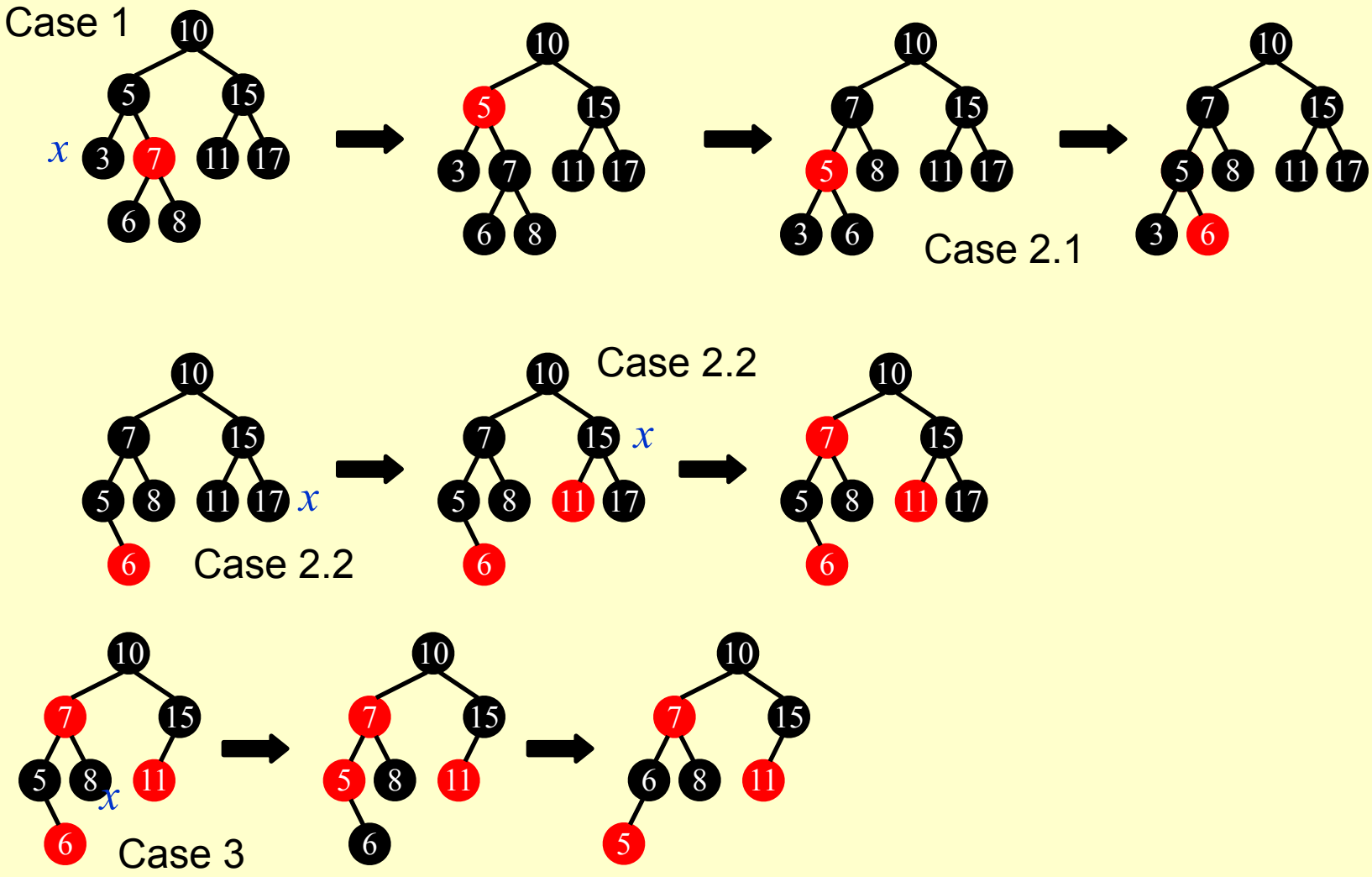

Case 1
x
Case 2.1
Case 2.2
x
Case 2.2

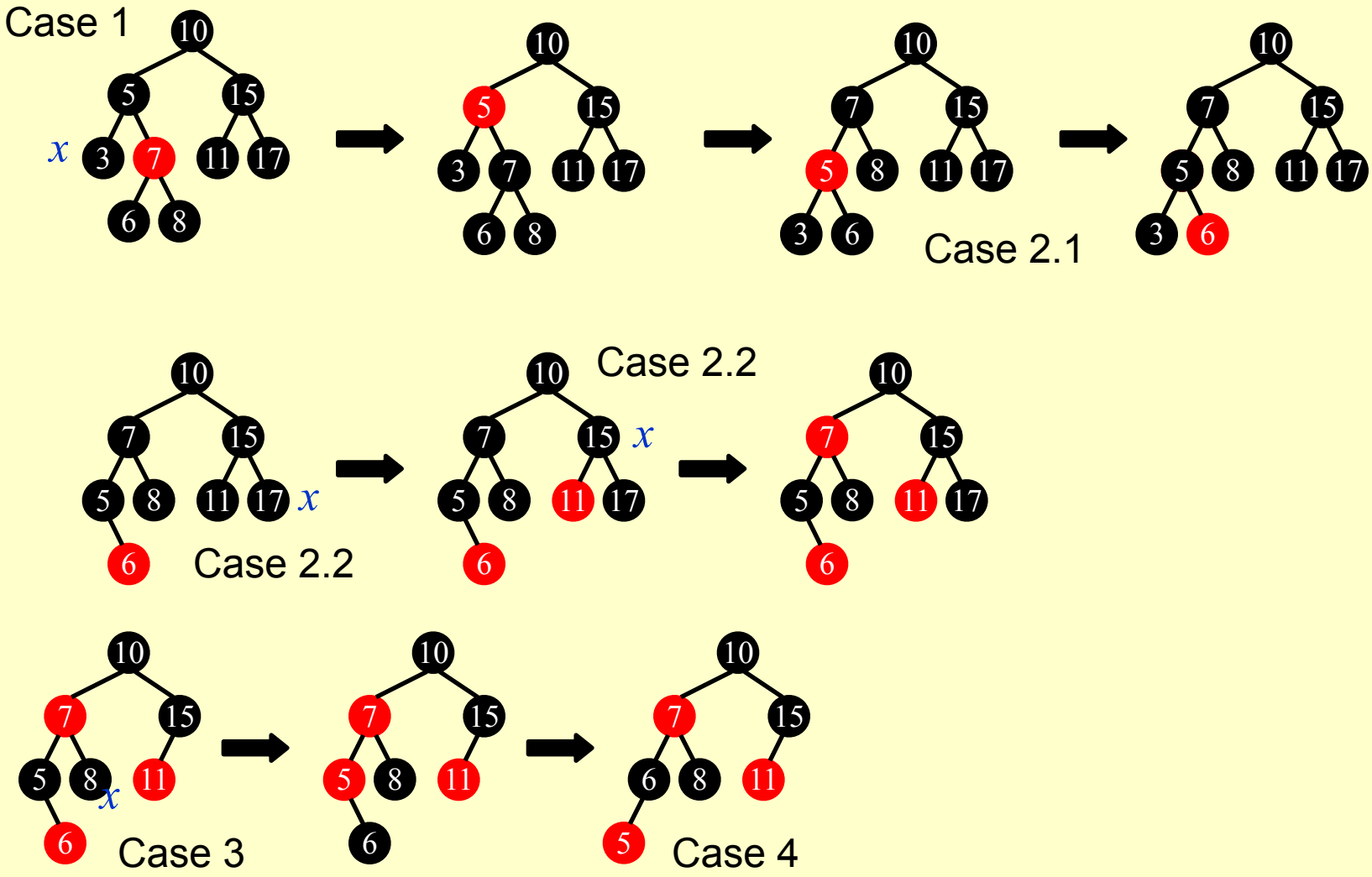

Case 1
x
Case 2.1
Case 2.2
x
Case 2.2

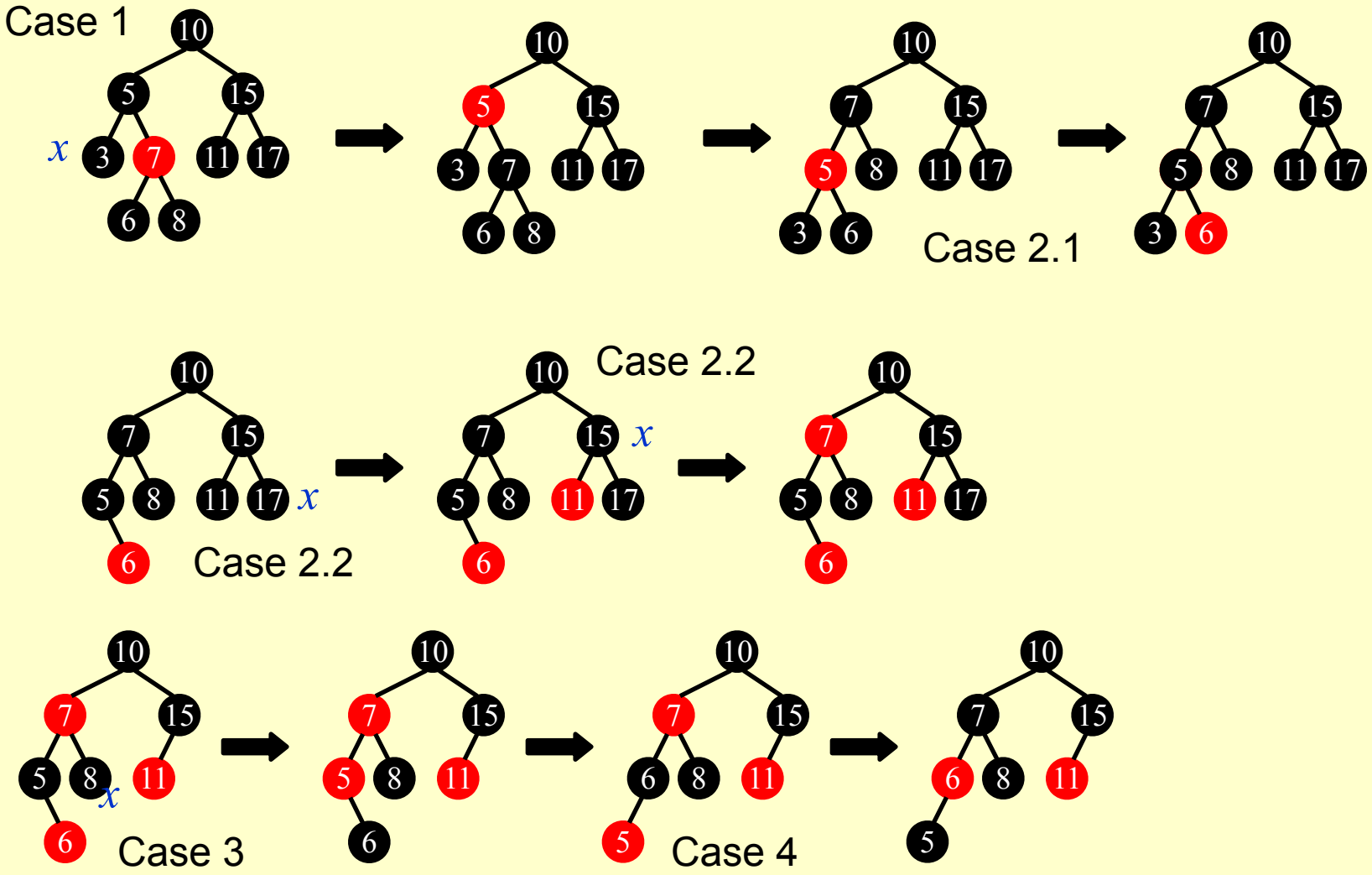

Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x

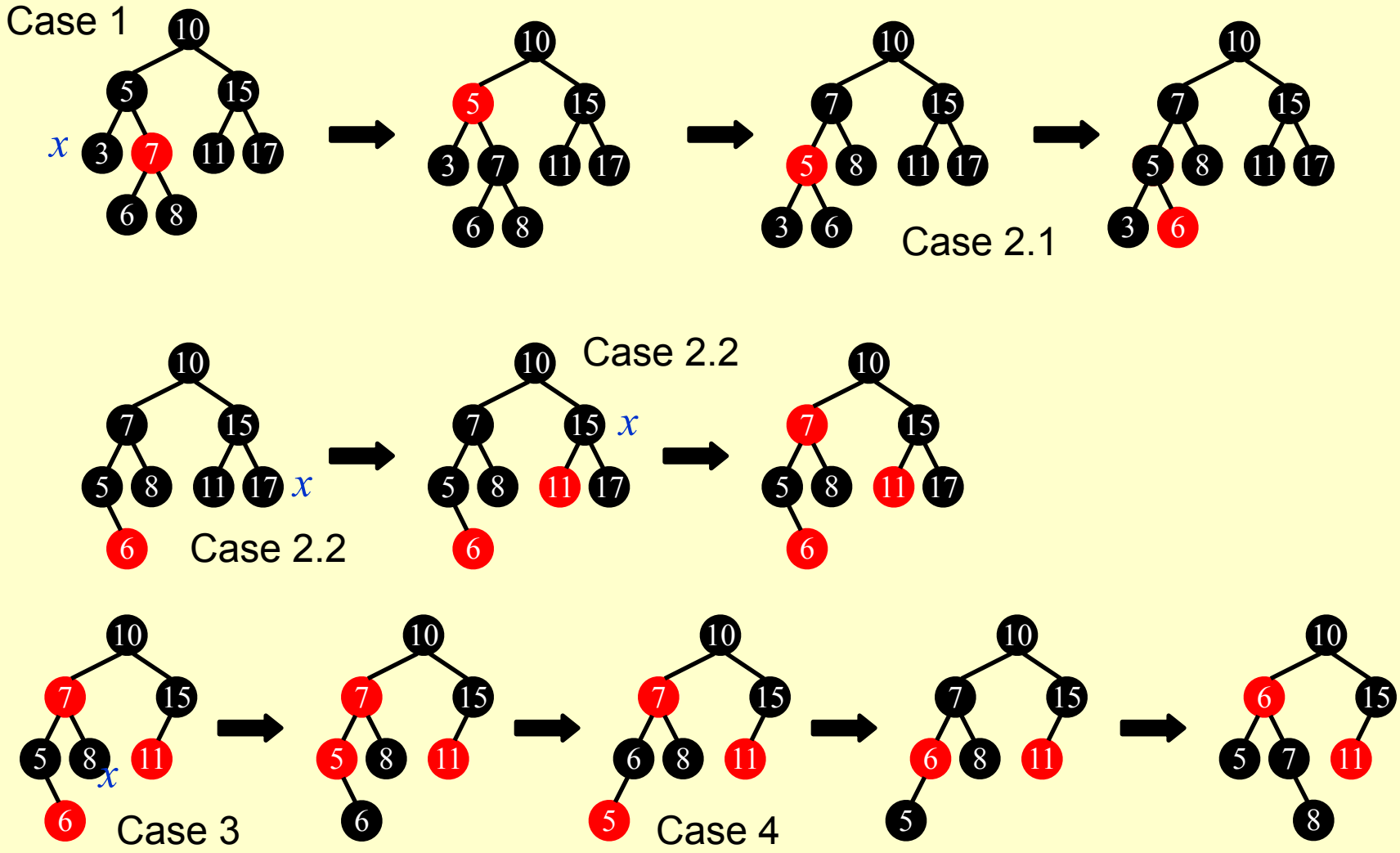

Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x
Case 3


Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x
Case 3


Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x
Case 3


Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x
Case 3
Case 4


Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x
Case 3
Case 4

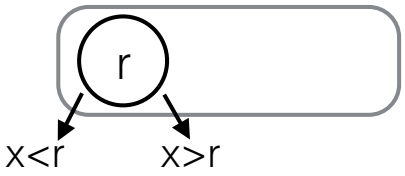

Case 1
x
Case 2.1
Case 2.2
x
Case 2.2
x
Case 3
Case 4

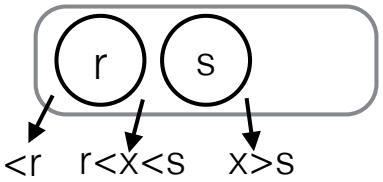
# Number of *rotations*

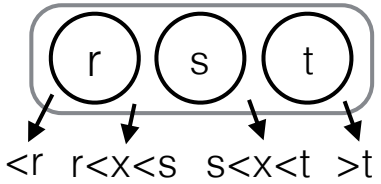
| | AVL | Red-Black Tree |
|---|---|---|
| Insertion | ≤ 2 | ≤ 2 |
| Deletion | $O(\log N)$ | ≤ 3 |

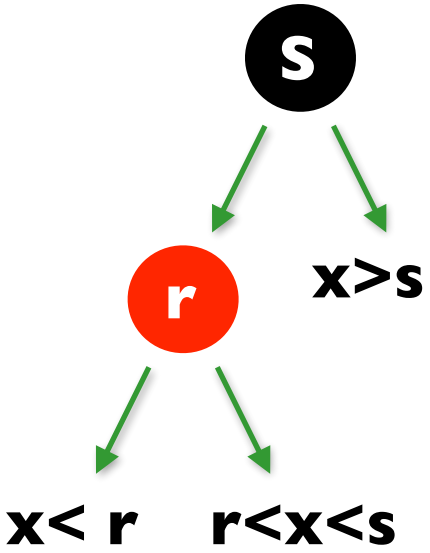
# Red-Black Trees

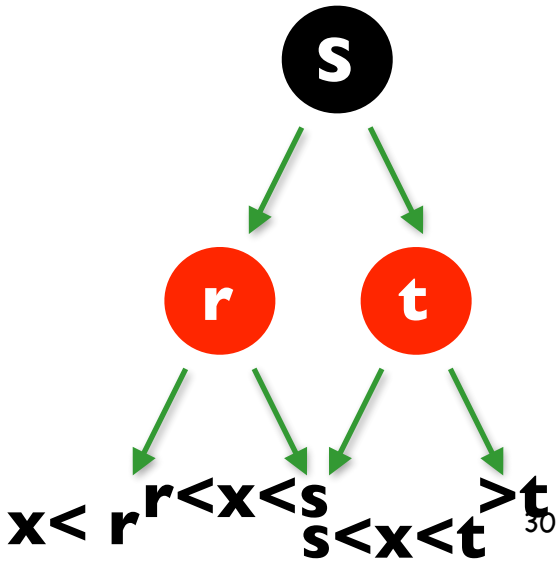
- Reduce 2-3-4 trees to BSTs:
  - The key is to transform 3- and 4- nodes into 2-nodes:



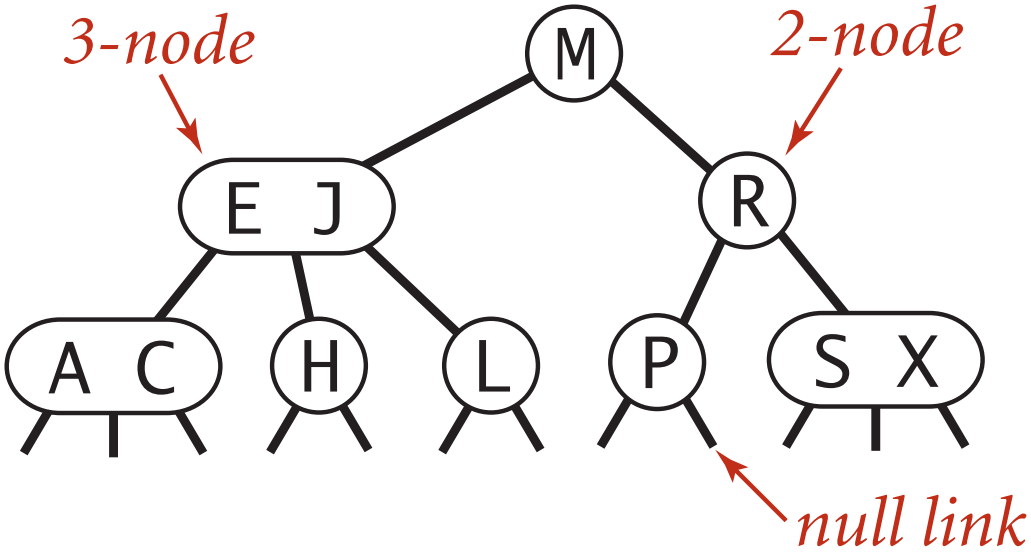



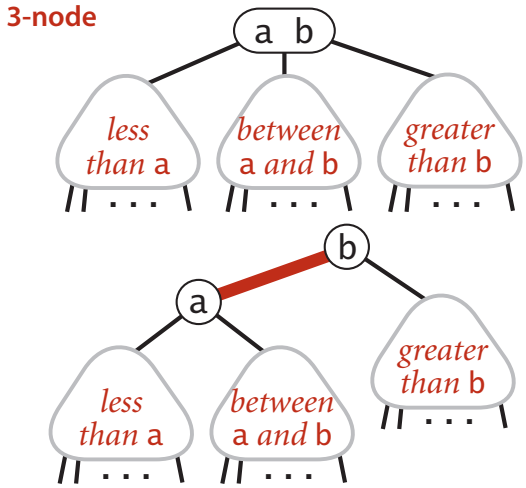



# 2-3 Trees

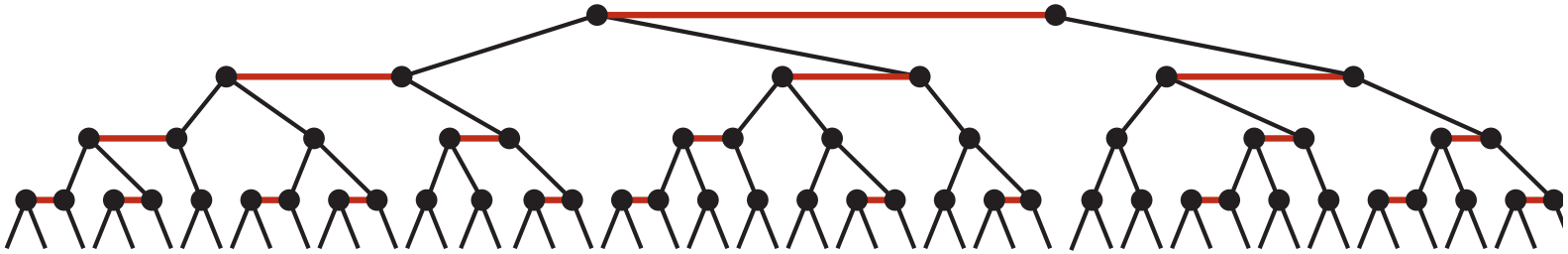



Transform into red-black tree?

# Left-Leaning Red-Black Trees




Encoding a 3-node with two 2-nodes connected by a left-leaning red link



A red-black tree with horizontal red links is a 2-3 tree

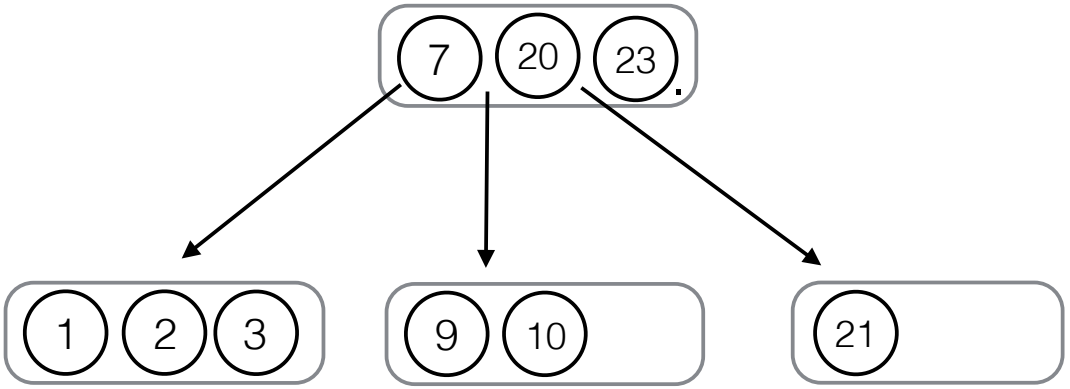
See [Sedgewick & Wayne] Chap. 3.3

# Balanced Search Trees (II)

- Review of amortized analysis
- Red-black trees
- B-trees
- Take-home messages

# M-ary Search Tree

- We can generalize binary search trees to M-ary search trees.

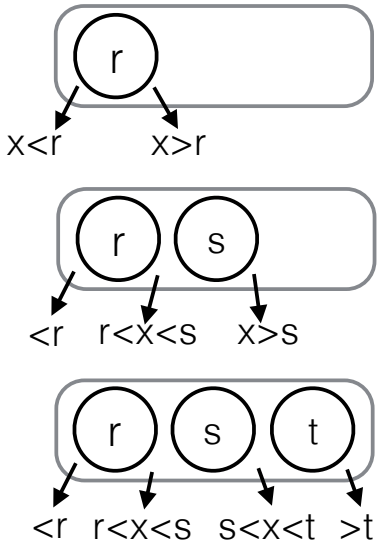



4-ary search tree:
Nodes have 1,2, or 3 data items and 0 to 4 children.

# 2-3-4 Trees

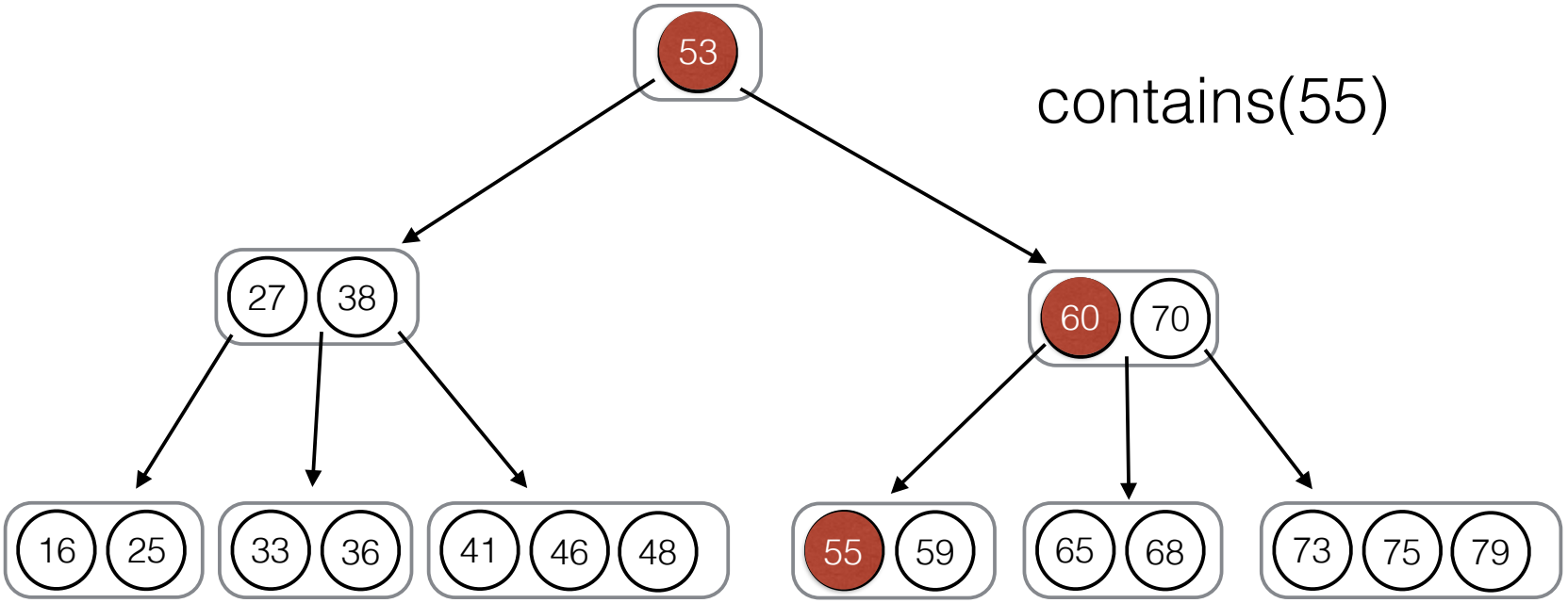
- A 2-3-4 Tree is a balanced 4-Ary search tree.

- Three types of internal nodes:
  - a 2-node has 1 item and 2 children.
  - a 3-node has 2 item and 3 children.
  - a 4-node has 3 item and 4 children.




- Balance condition:
  All leaves have the same depth.
  (height of the left and right subtree is always identical)

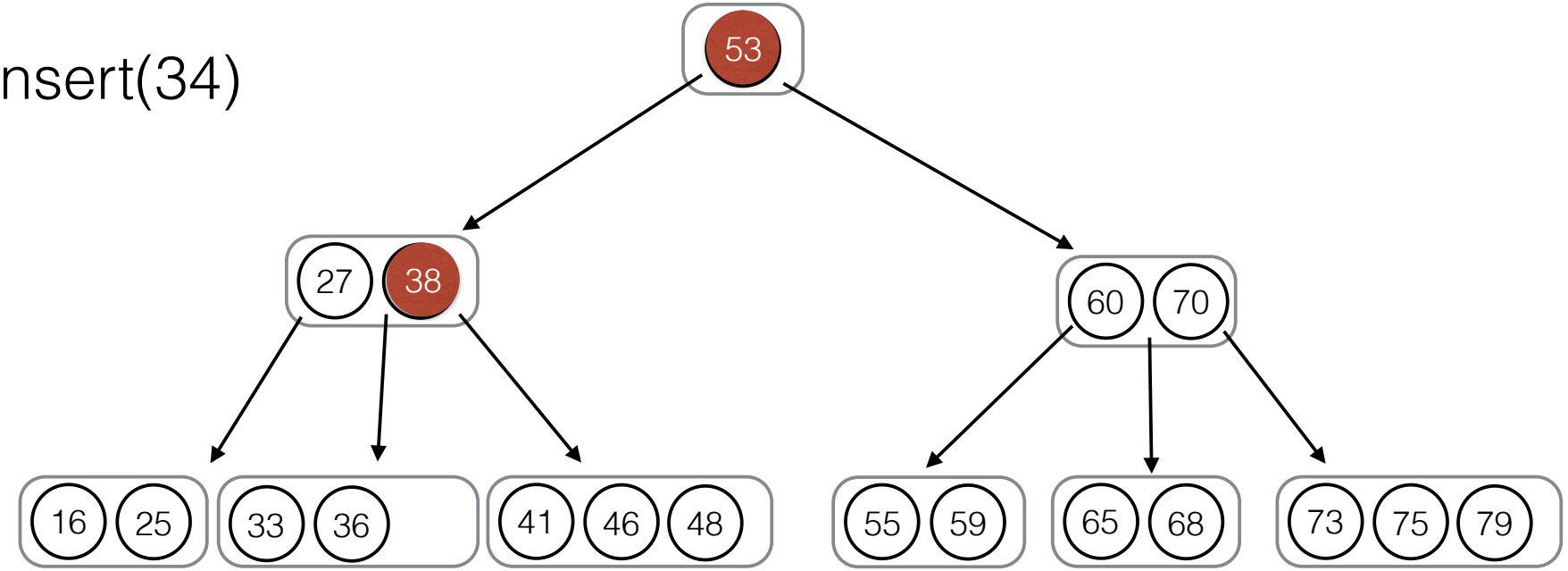
# `contains` in a 2-3-4 Tree




- At each level try to find the item: 2 steps = O(c)
- If not found, follow reference down the tree. There are at most O(height(T)) = O(log N) references.

# `insert` into a 2-3-4 Tree

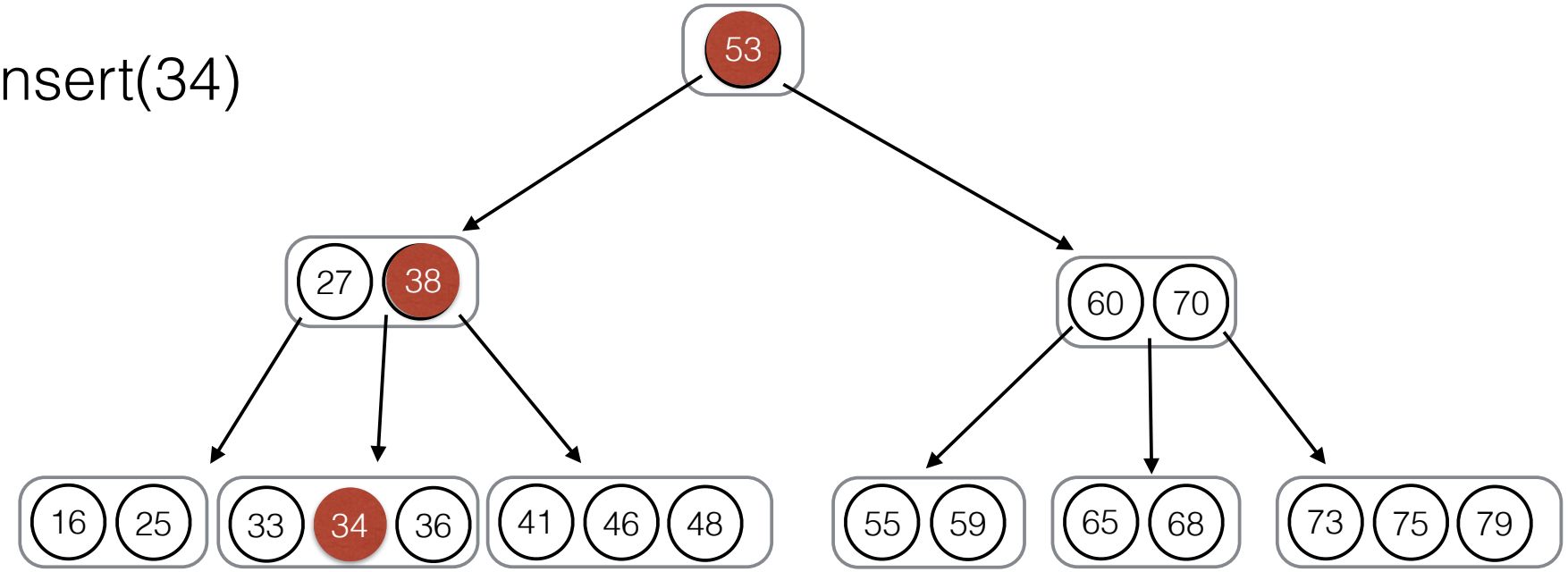
insert(34)




- Follow the same steps as contains.
- If X is found, do nothing.
- If there is still space in the leaf that should contain X, add it.

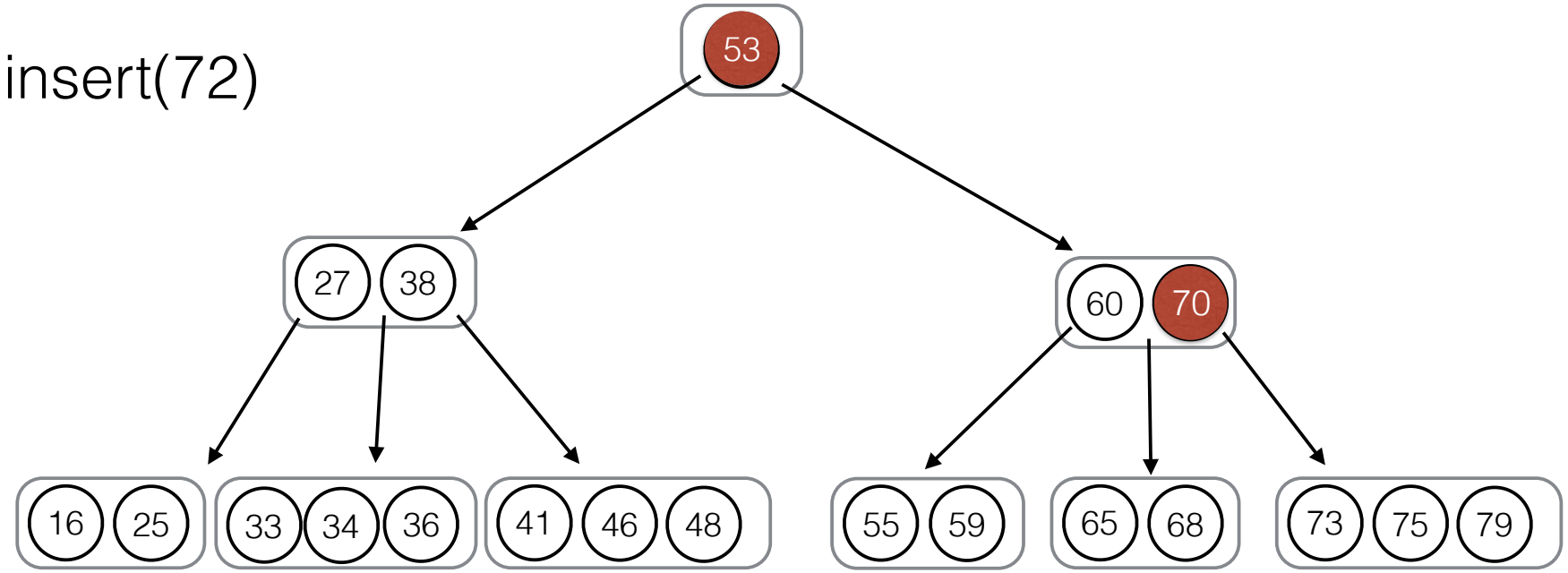
# `insert` into a 2-3-4 Tree

insert(34)



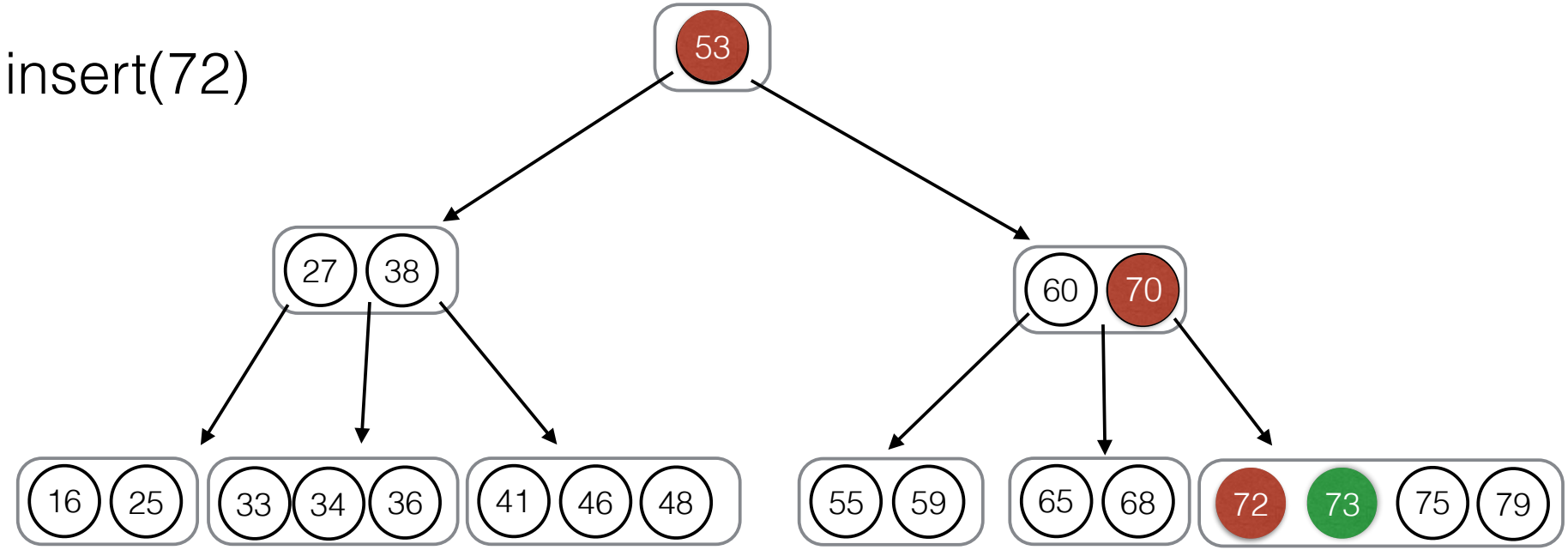

- Follow the same steps as contains.
- If X is found, do nothing.
- If there is still space in the leaf that should contain X, add it.
- **What if the leaf is full?**

# `insert:` splitting nodes



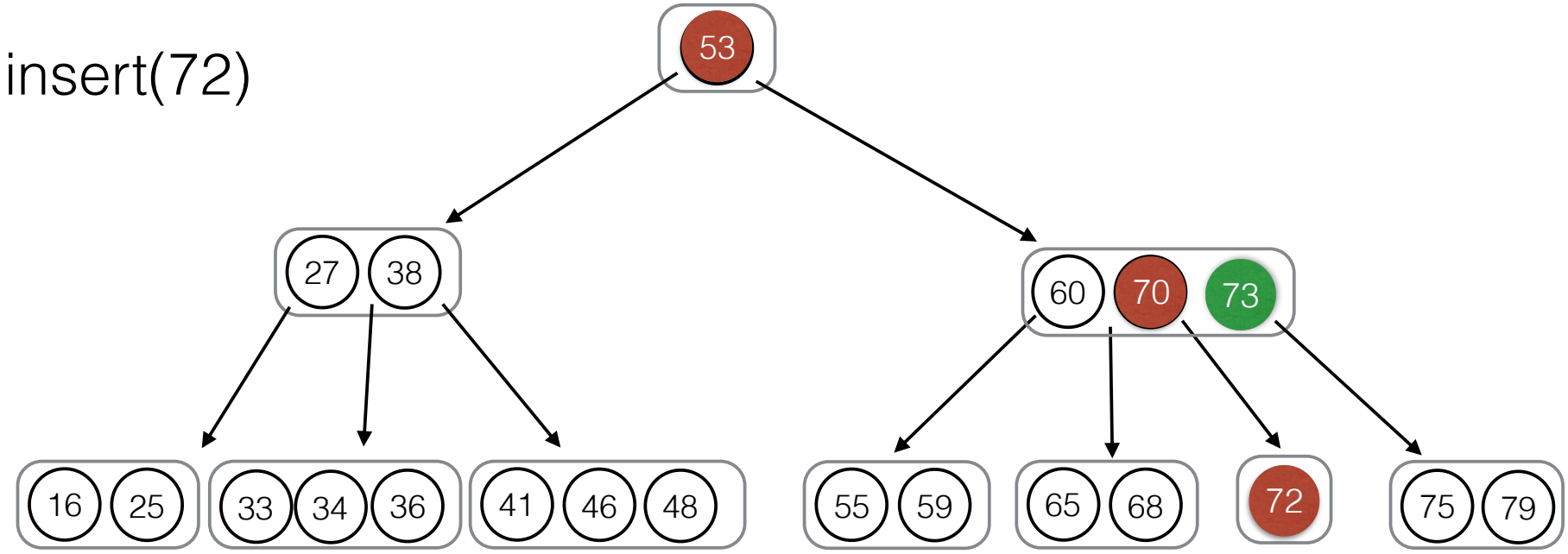

- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

# `insert`: splitting nodes




- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

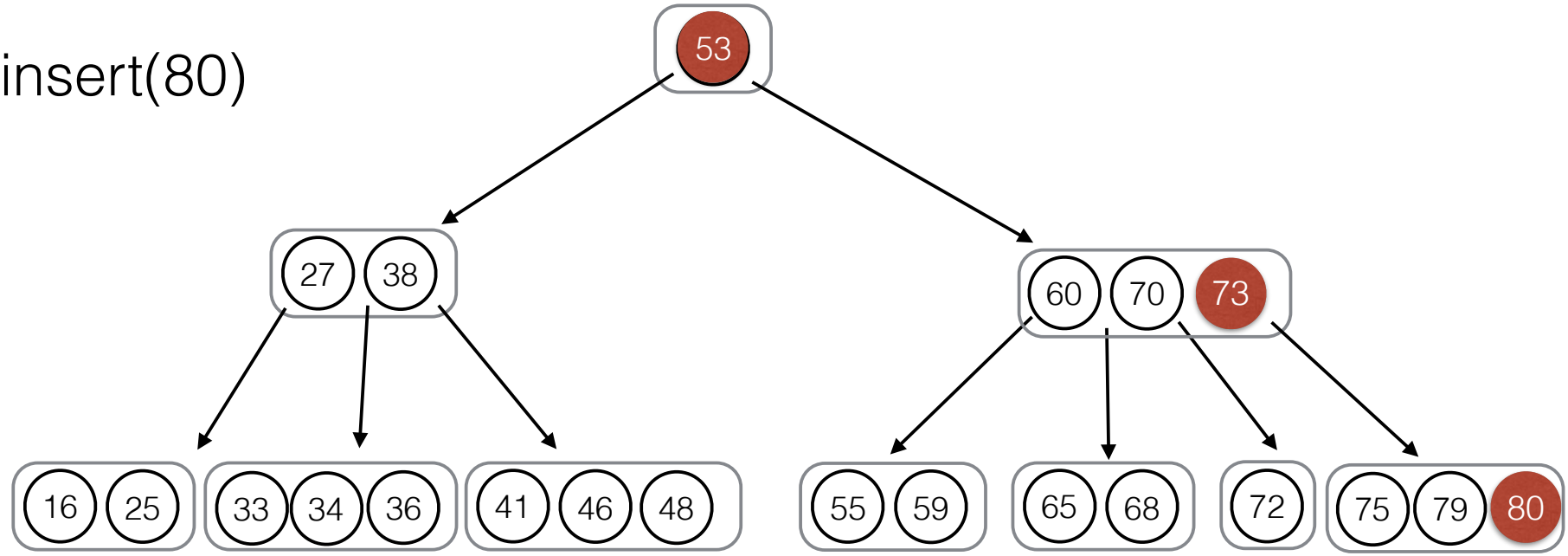
# `insert`: splitting nodes



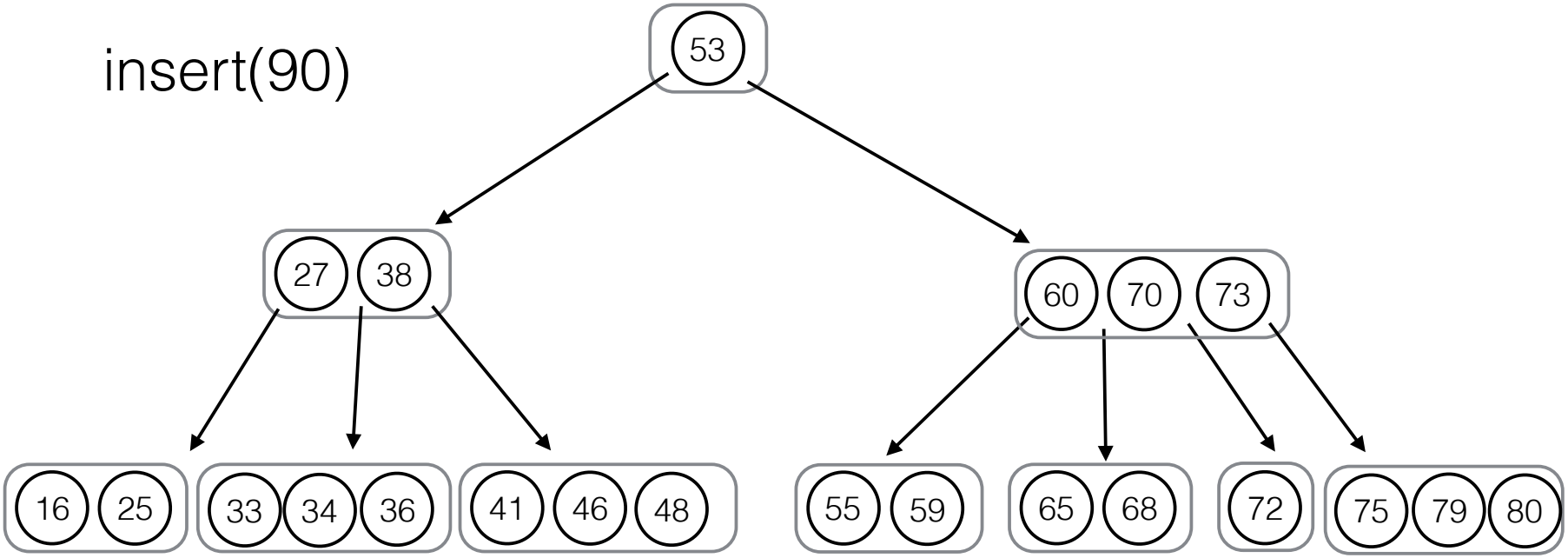

- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

# `insert`: splitting nodes

insert(80)



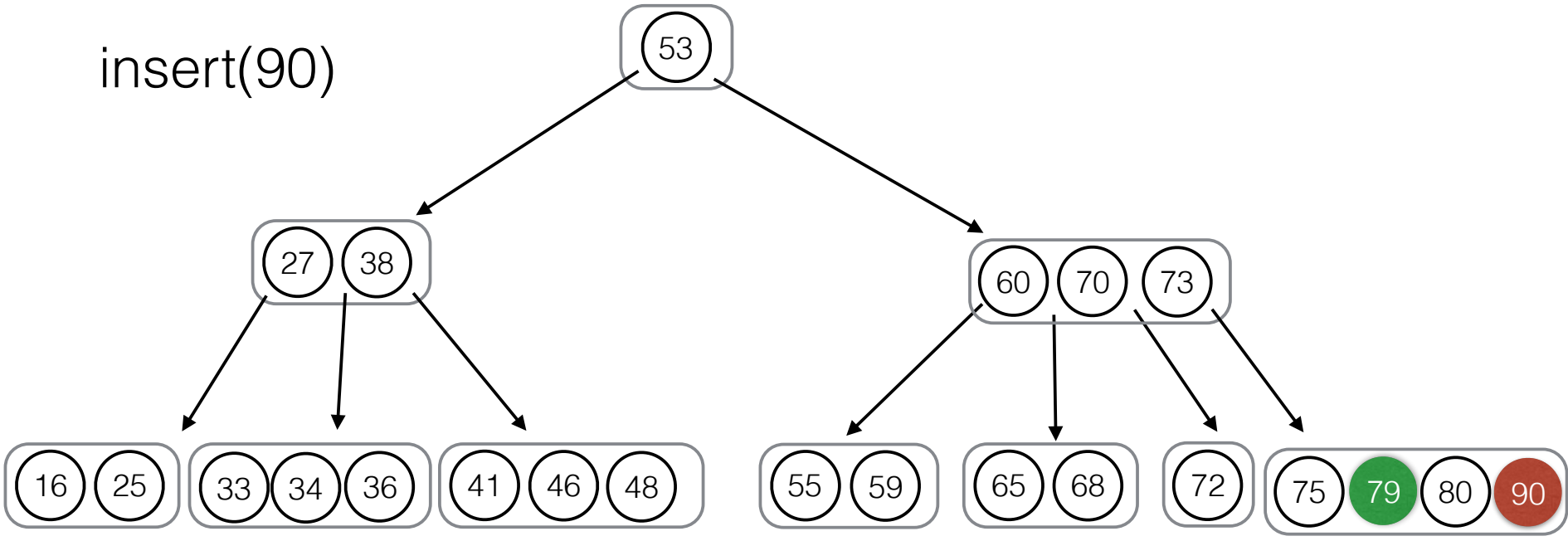
# `insert`: splitting nodes




- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with ~~old root as a single child~~.
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

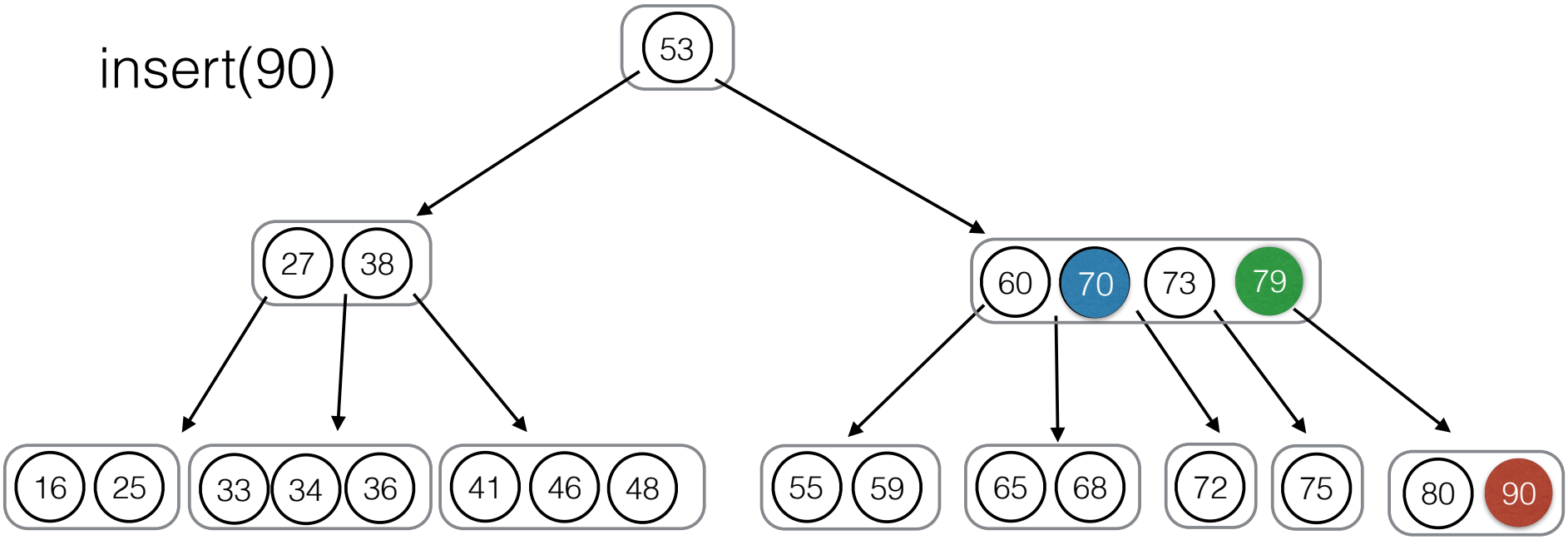
**splitting old root as two children**

# `insert`: splitting nodes



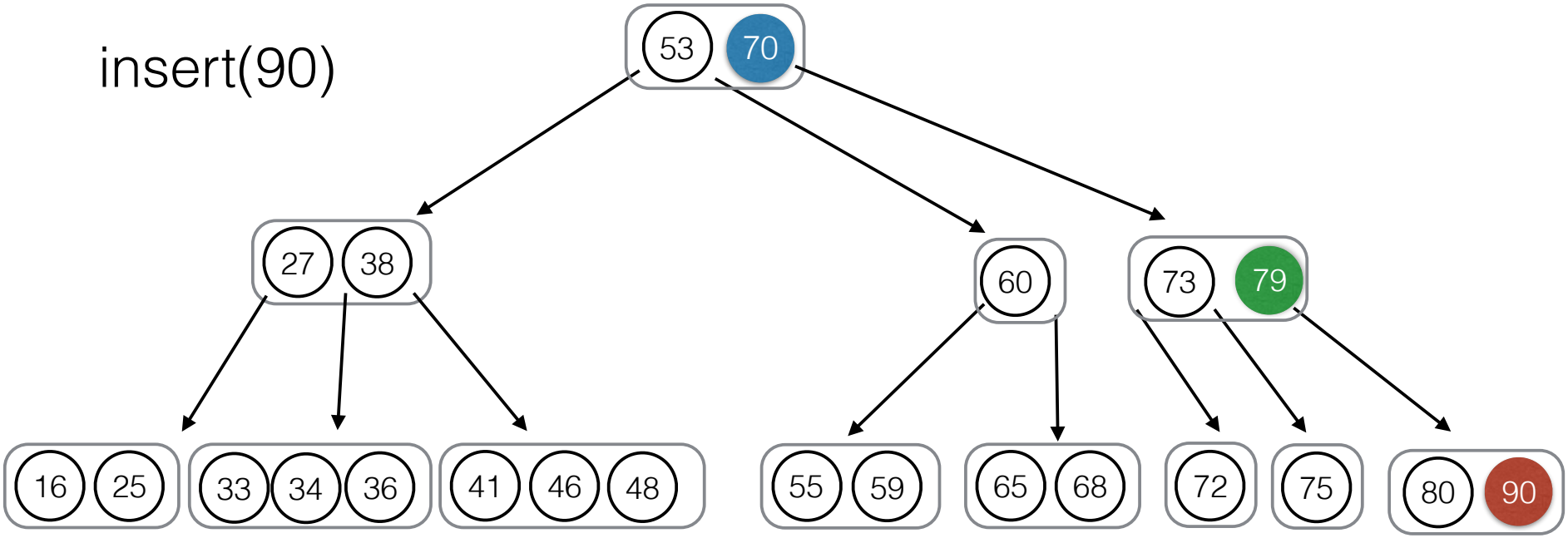

- If parent is also full, continue to split the parent until space can be found. **splitting old root as two children**
- If root is full, create a new root with ~~old root as a single child~~.
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

# `insert`: splitting nodes



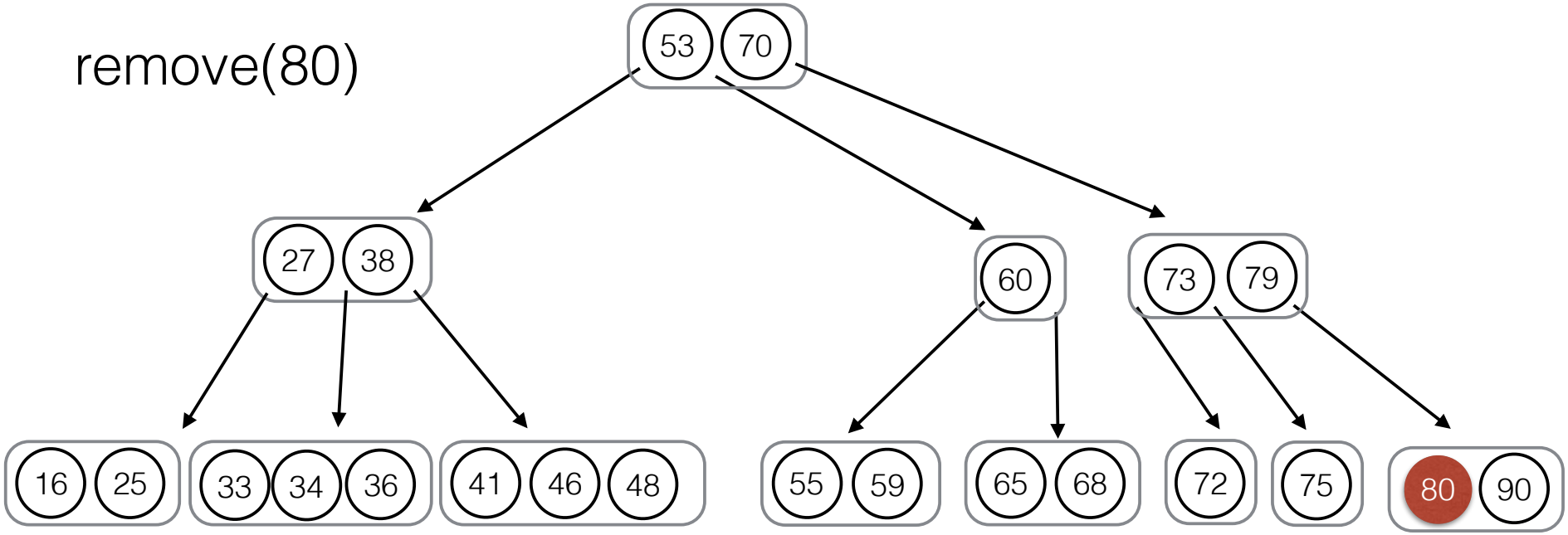

- If parent is also full, continue to split the parent until space can be found. **splitting old root as two children**
- If root is full, create a new root with ~~old root as a single child~~.
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

# `insert`: splitting nodes



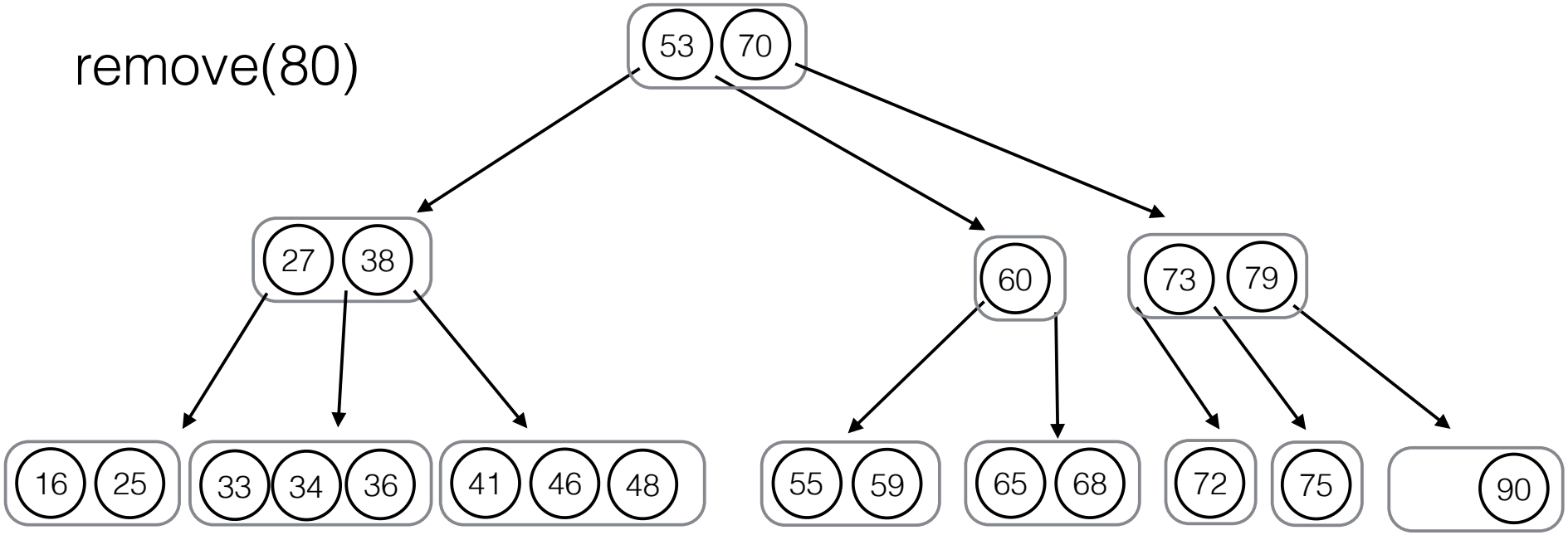

- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with ~~old root as a single child~~. **splitting old root as two children**
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

# `remove` from a 2-3-4 tree



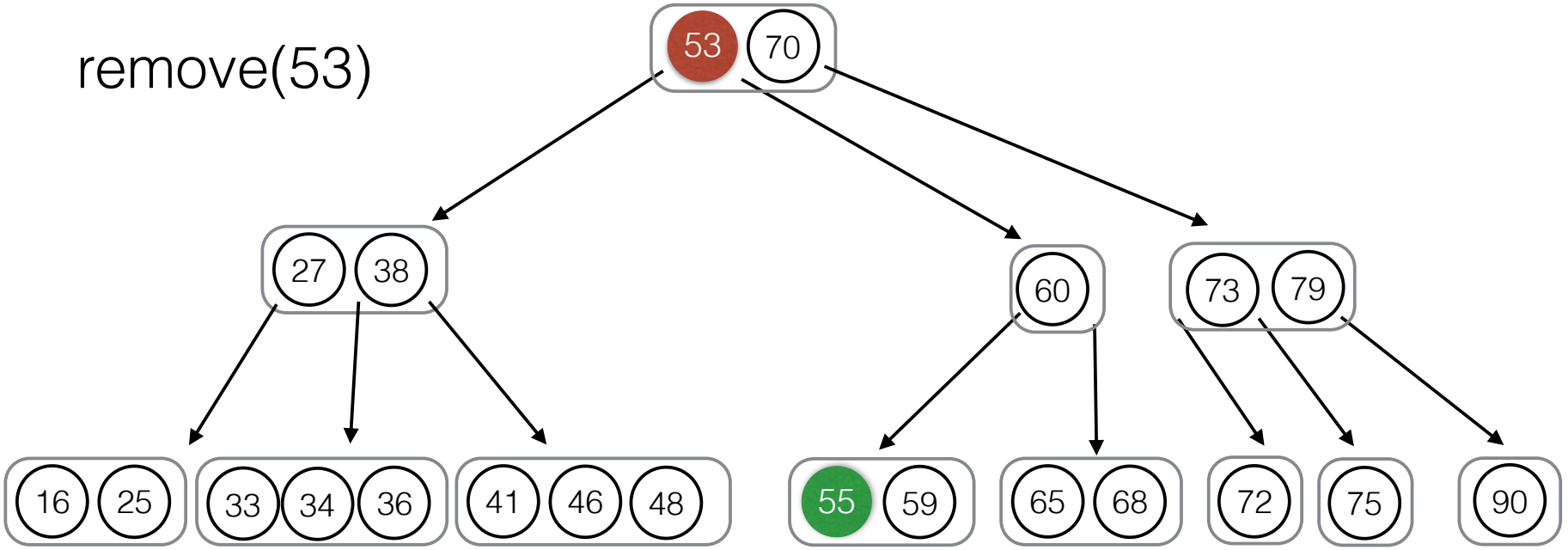

- Item in a 3- or 4-leaf can just be removed.

# `remove` from a 2-3-4 tree



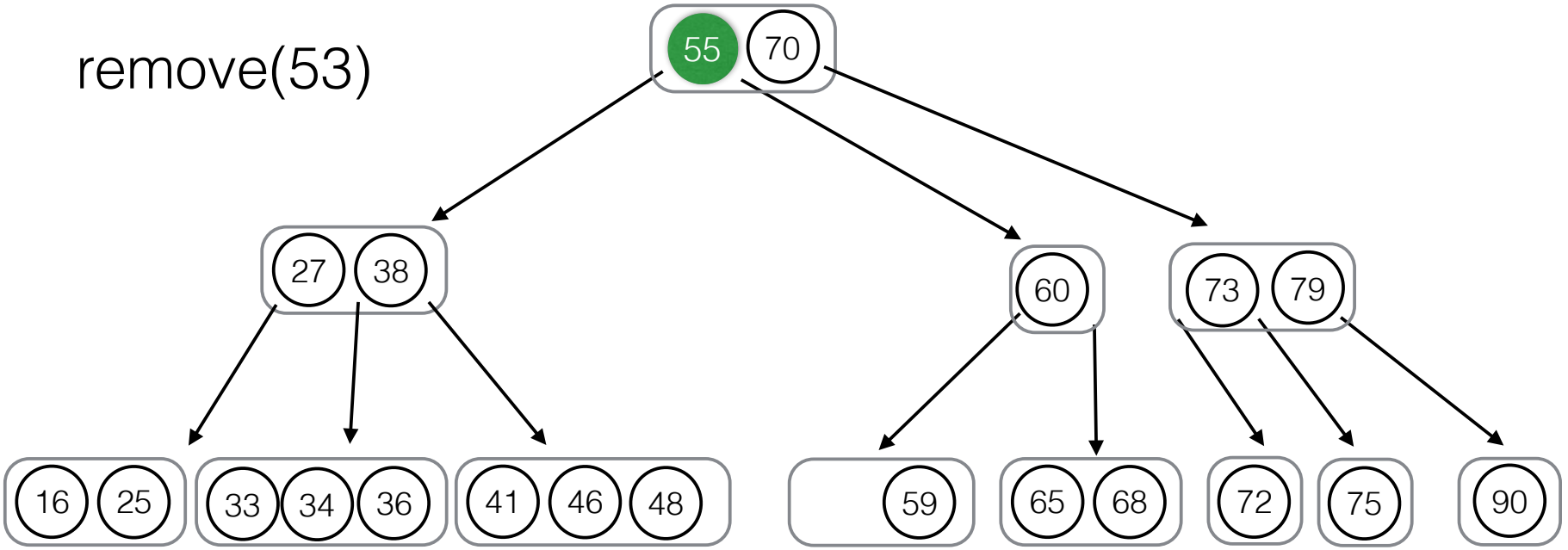

- Item in a 3- or 4-leaf can just be removed.

# `remove` from a 2-3-4 tree



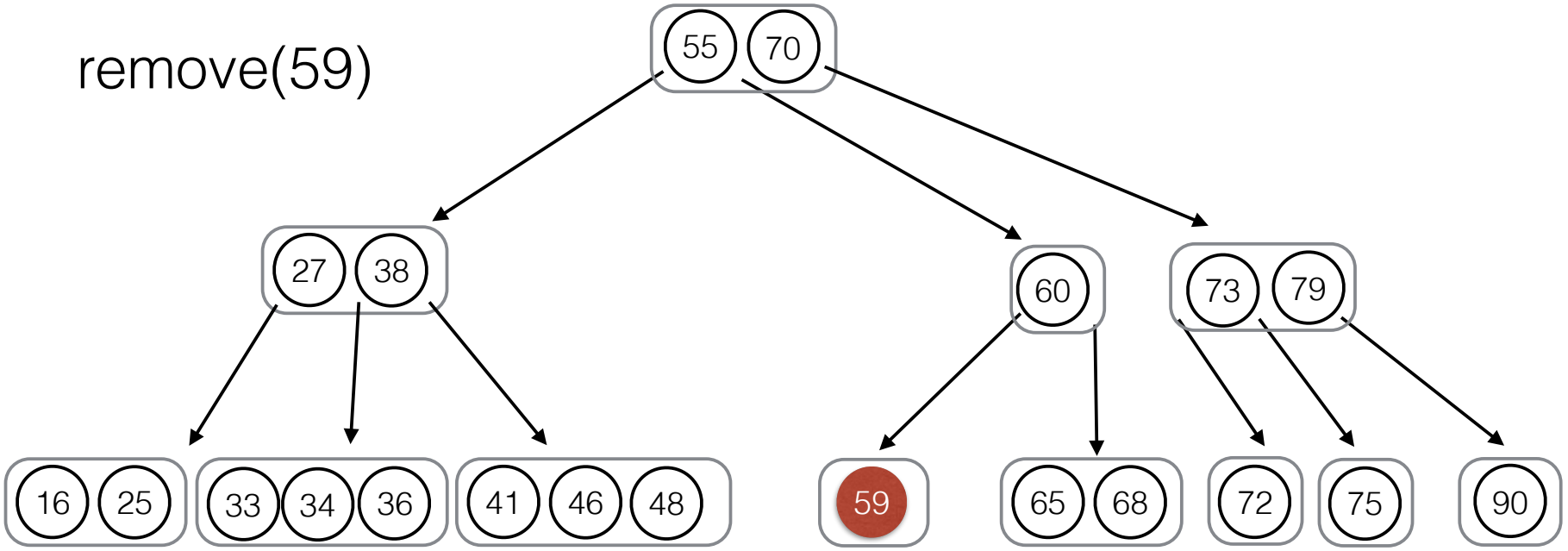

- Removal of an item v from internal node:
  - Continue down the tree to find the leaf with the next highest item *w*. Replace *v* with *w*. Remove *w* from its original position recursively.

# `remove` from a 2-3-4 tree



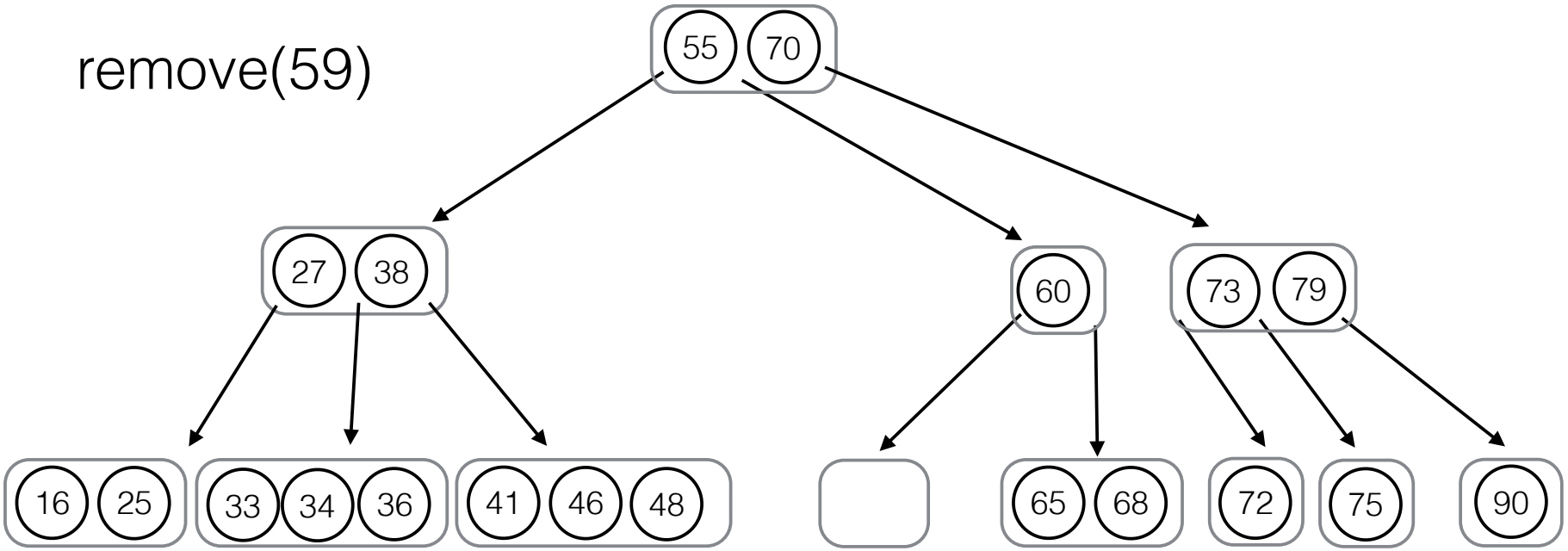

- Removal of an item v from internal node:
  - Continue down the tree to find the leaf with the next highest item *w*. Replace *v* with *w*. Remove *w* from its original position recursively.

# `remove` from a 2-3-4 tree



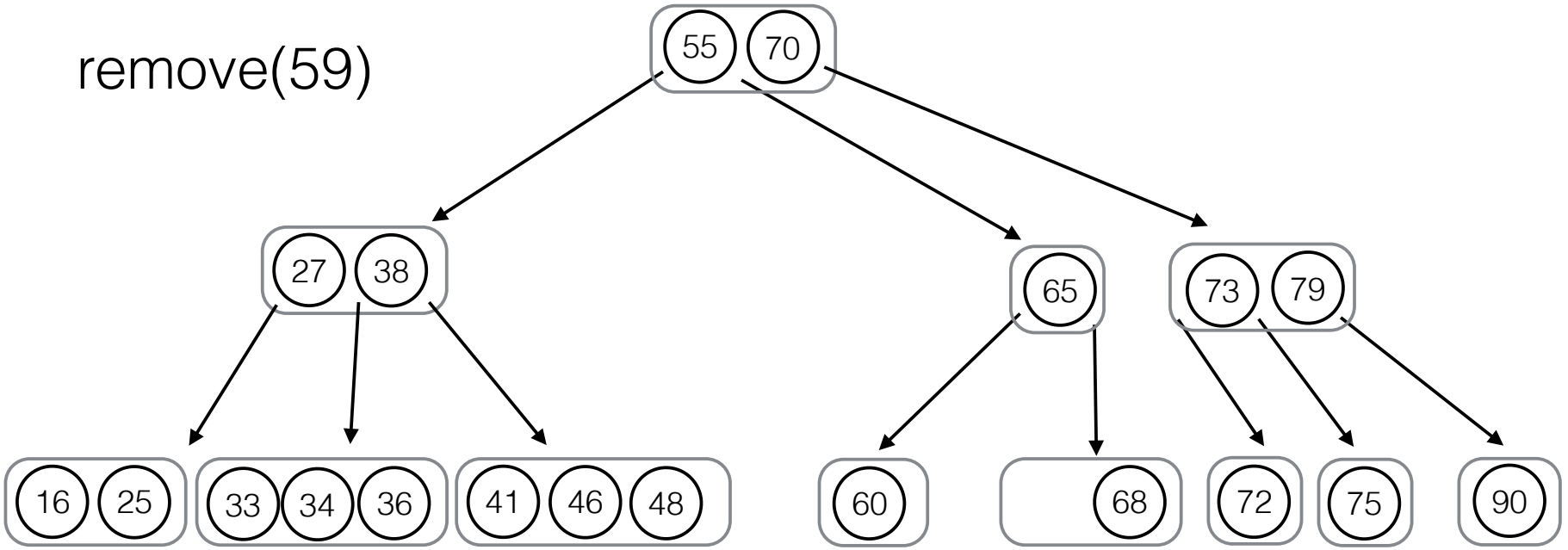

- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

# `remove` from a 2-3-4 tree




- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

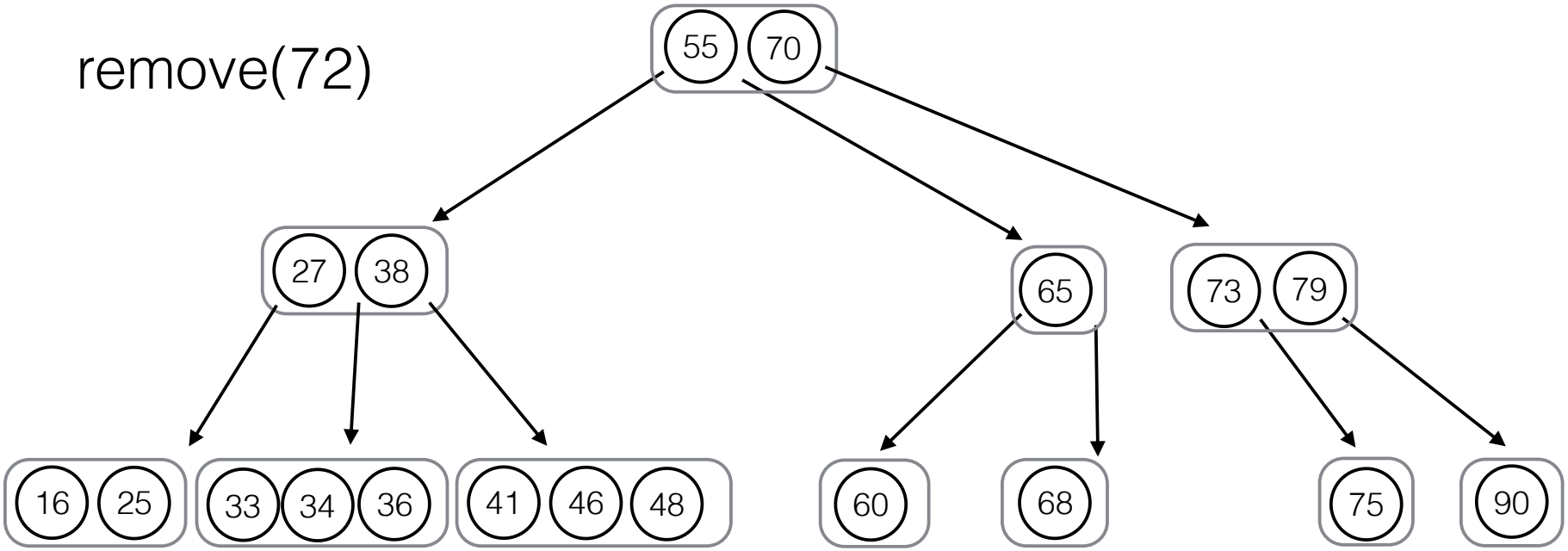
# `remove` from a 2-3-4 tree




- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

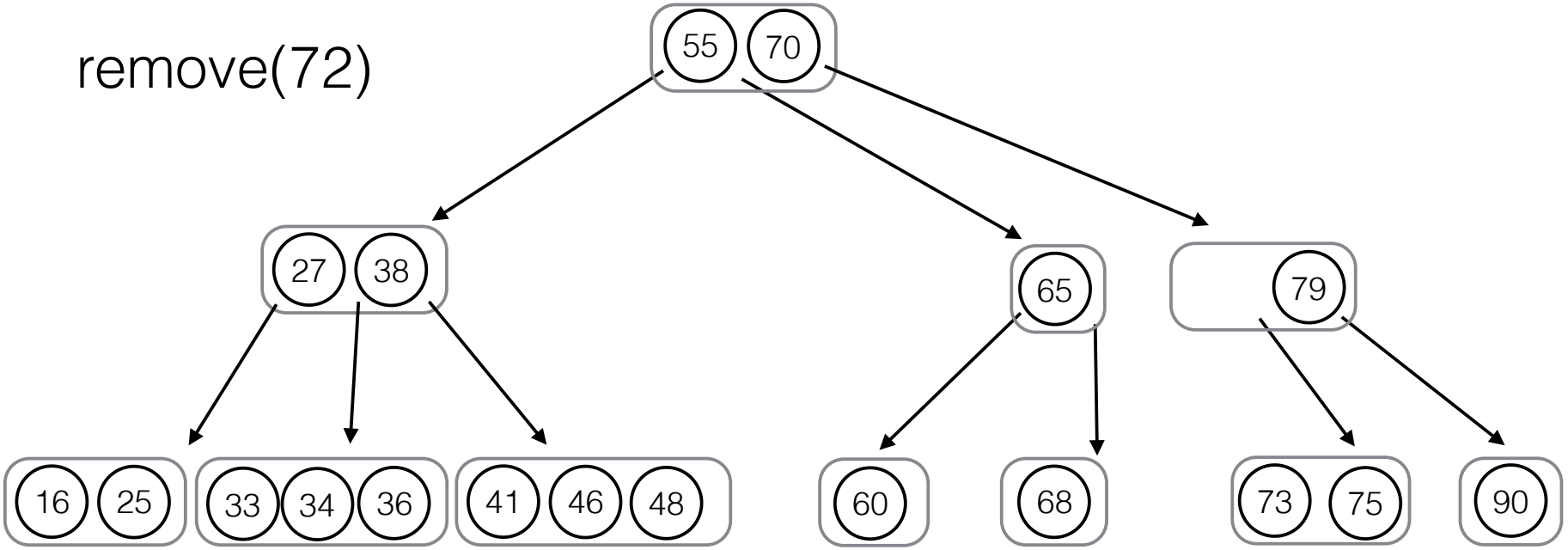
**What if no sibling is a 3 or 4 node?**

# `remove` from a 2-3-4 tree



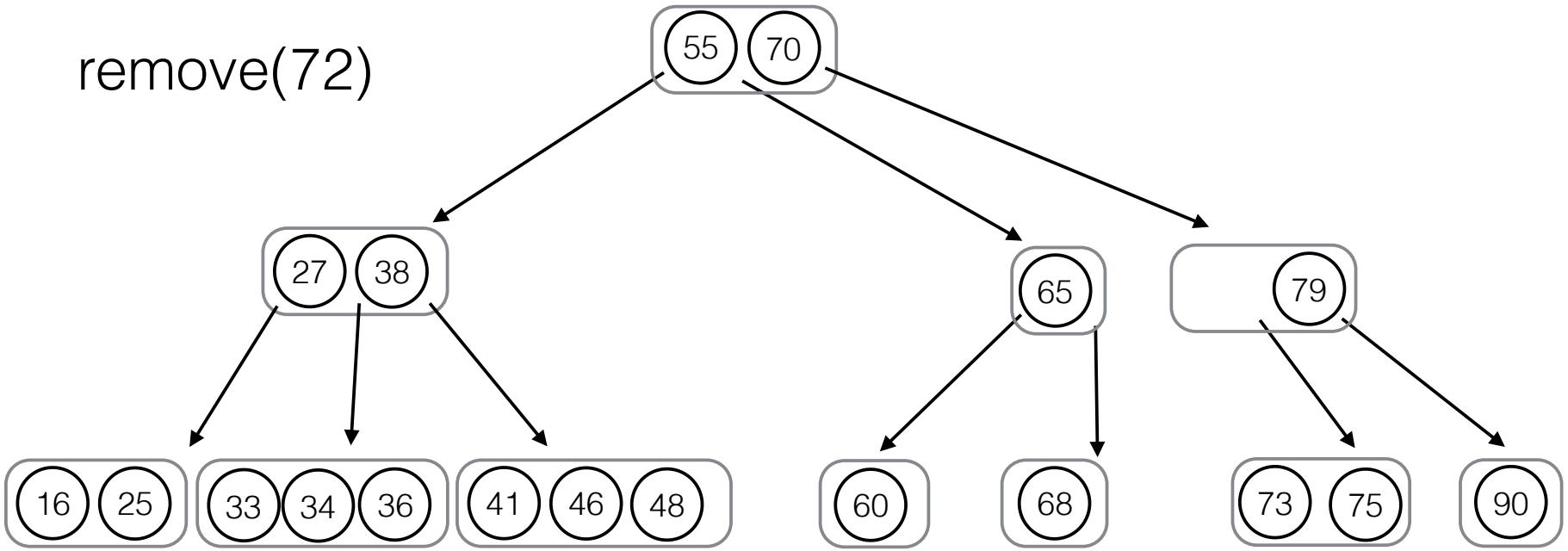

- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

# `remove` from a 2-3-4 tree




- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

# `remove` from a 2-3-4 tree

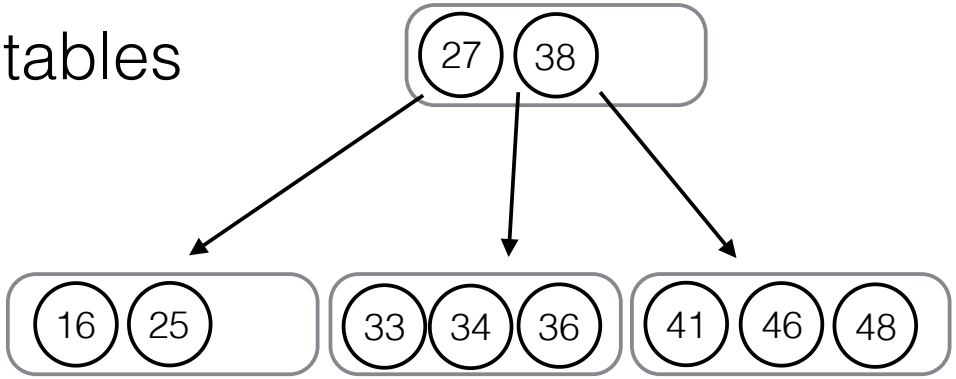



- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

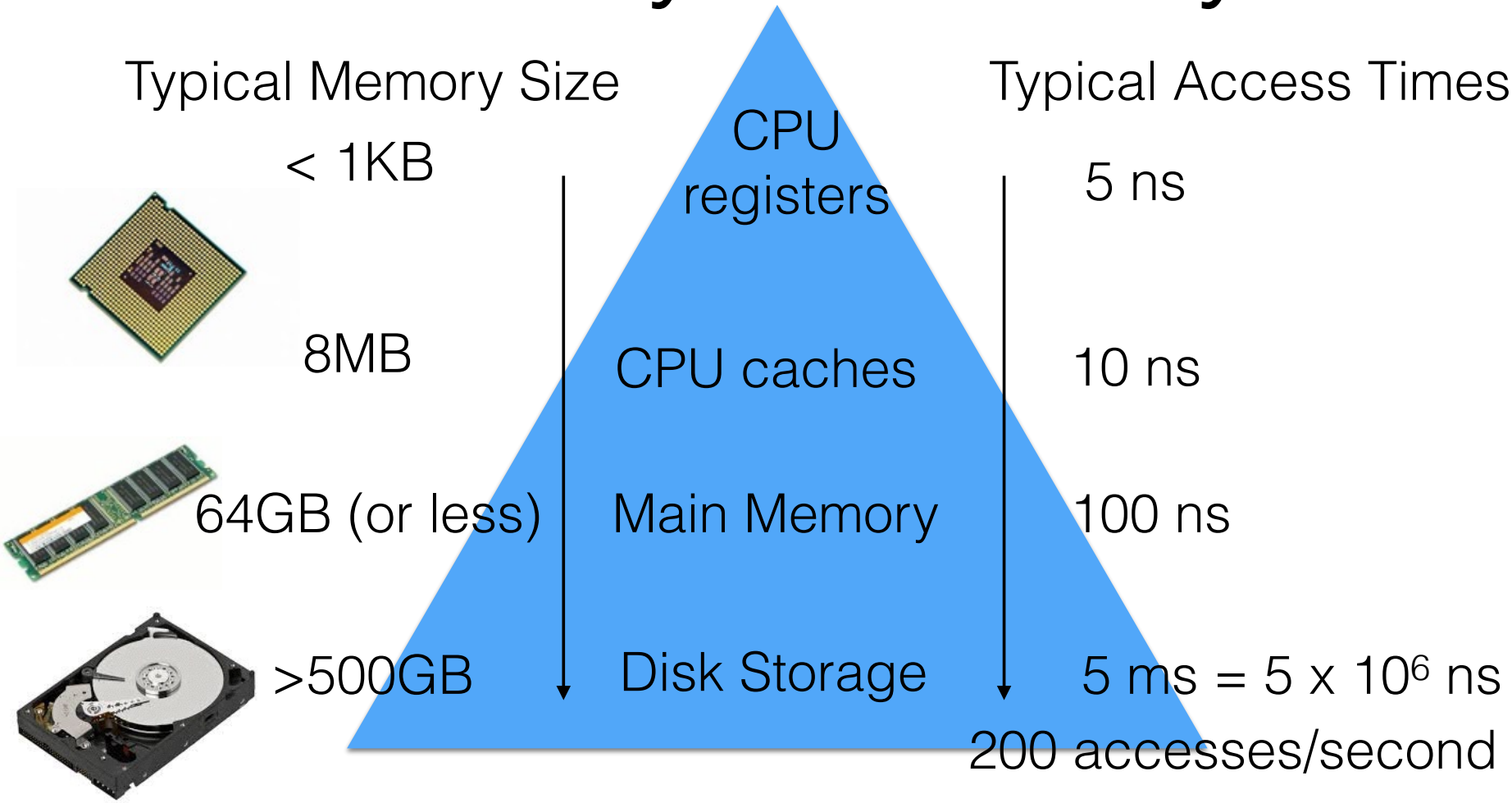
All modifications to fix the tree are local and therefore O(c).
Remove runs in O(log N).

# B-Trees

- A B-Tree is a generalization of the 2-3-4 tree to M-ary search trees.
- Every internal node (except for the root) has $\lceil \frac{M}{2} \rceil \leq d \leq M$ children and contains $d-1$ values.
- All leaves contain $\lceil \frac{L}{2} \rceil \leq d \leq L$ values (usually L=M-1)
- All leaves have the same depth.
- Often used to store large tables on hard disk drives. (databases, file systems)



# Memory Hierarchy




Memory access is **much** faster than disk access.

# Large BST on Disk (1)

- Assume we have a very large database table, represented as a binary search tree:
  - 10 million items, 256 bytes each.
  - 6 disk accesses per second (shared system).
- Assume no caching, every lookup requires disk access.

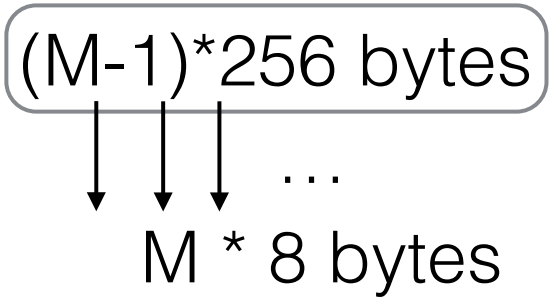
# Large BST on Disk (2)

- Disk access time for finding a node in an unbalanced BST:
  - depth of searched node is *N* in the **worst case**:
    - 10 million items -> 10 million disk accesses
    - 10 million / 6 accesses per second ≈ 19 days!
  - **Expected** depth is *1.38 log N*
    - *1.38 $\log_2$ 10 x $10^6$ items ≈ 32 disk accesses*
    - *32 / 6 accesses per second ≈ 5 seconds*

# Large BST on Disk (2)

- Even for AVL Tree the worst case and average case will be around log N.
- *A*bout 24 disk accesses in 4 sec.

# Estimating the ideal M for a B-Tree




- Assume 8KB= 8,192 byte block size.
- Every data item is 256 byte.
- An M-ary B-Tree contains at most *M-1* data items + M block addresses of other trees (a 8 byte pointer each).
- How big can we make the nodes?

$$(M-1)\cdot 256\ \text{byte} + M \cdot 8\ \text{byte} = 8,192\ \text{byte}$$

$$M = 32$$

# Calculating Access Time

- We representing 10,000,000 items in a B-Tree with M=32

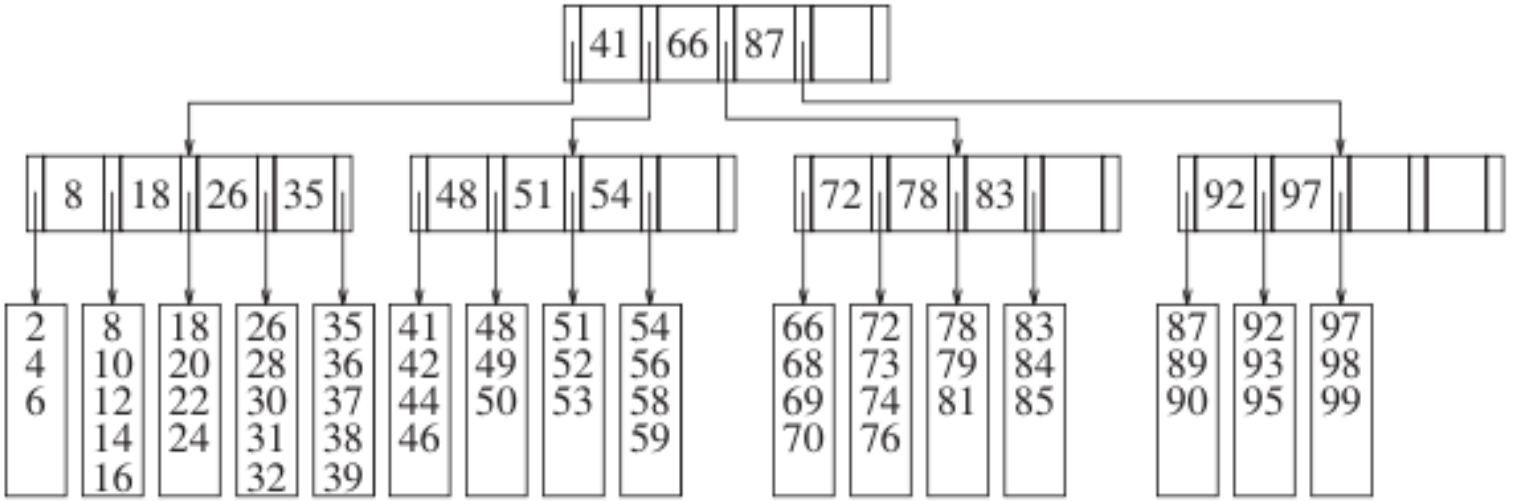
- The tree has a worst-case height of $log_{\frac{M}{2}} N$

$$log_{\frac{32}{2}} 10,000,000 \approx 6$$

- Worst-case time to find an item is
  6 accesses / 6 disk accesses per second = 1 *second*

# B+ Trees

- Only leafs store full (key, value) pairs.
- Internal nodes only contain keys to help find the right leaf.
- Insert/removal only at leafs (slightly simpler, see book).



# B+ Trees on Disk

- Assume keys are 32 bytes.

$$(M-1) \cdot 32 \text{ byte} + M \cdot 8 \text{ byte} = 8,192 \text{ byte}$$

- We can fit at most M=205 keys in each node.
- Worst case time for 1 million keys:

$$\log_{\frac{205}{2}} 10,000,000 = 3$$

  - 3 accesses / 6 seconds per access = .5 seconds

# B+ Trees

# B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

Assume each nonroot leaf also has between ⌈M/2⌉ and M children.

# B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

Assume each nonroot leaf also has between ⌈M/2⌉ and M children.

A B+ tree of order 4
(2-3-4 tree)

# B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

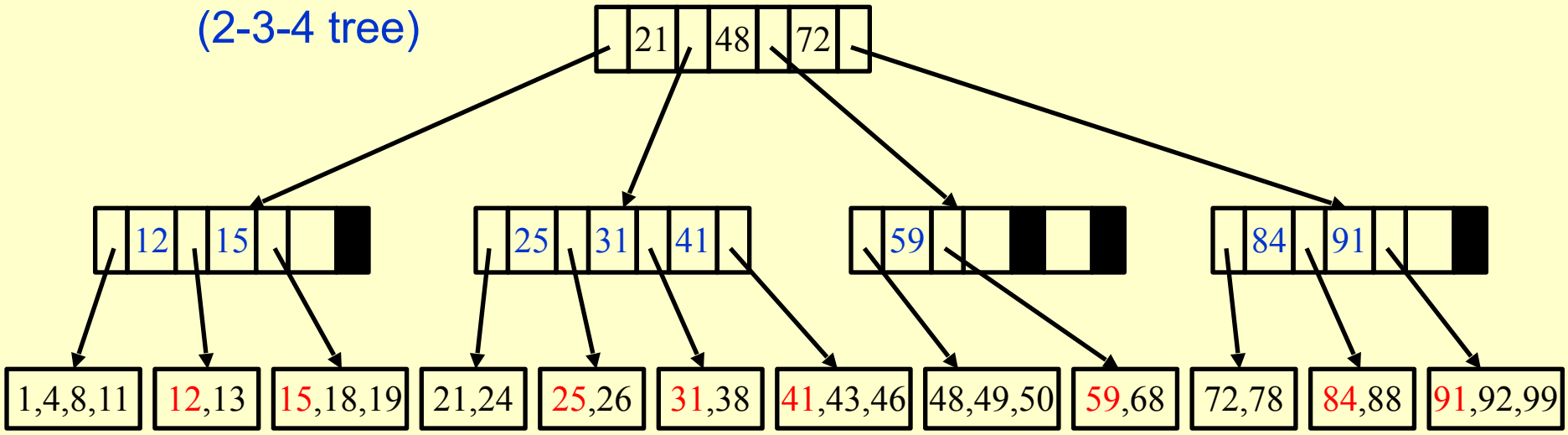
(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

Assume each nonroot leaf also has between ⌈M/2⌉ and M children.

A B+ tree of order 4
(2-3-4 tree)



# B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

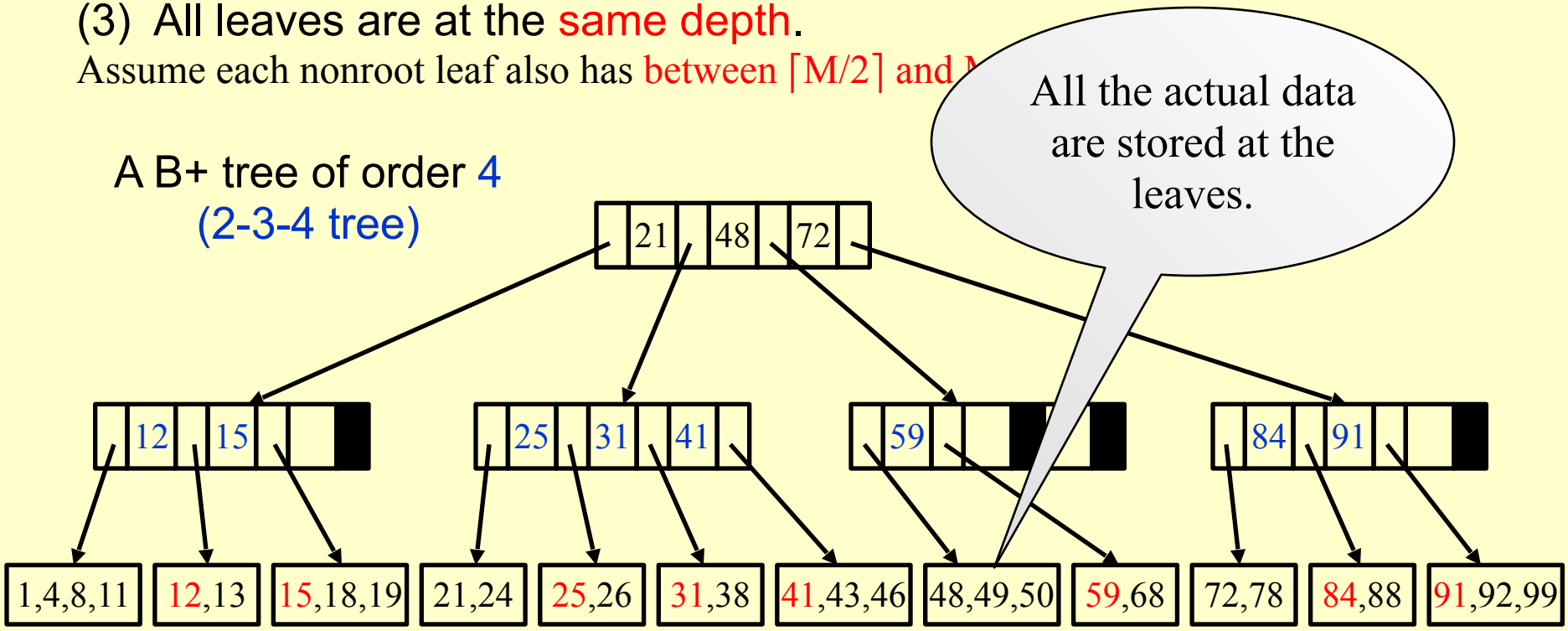
(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

Assume each nonroot leaf also has between ⌈M/2⌉ and M

A B+ tree of order 4
(2-3-4 tree)



# B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

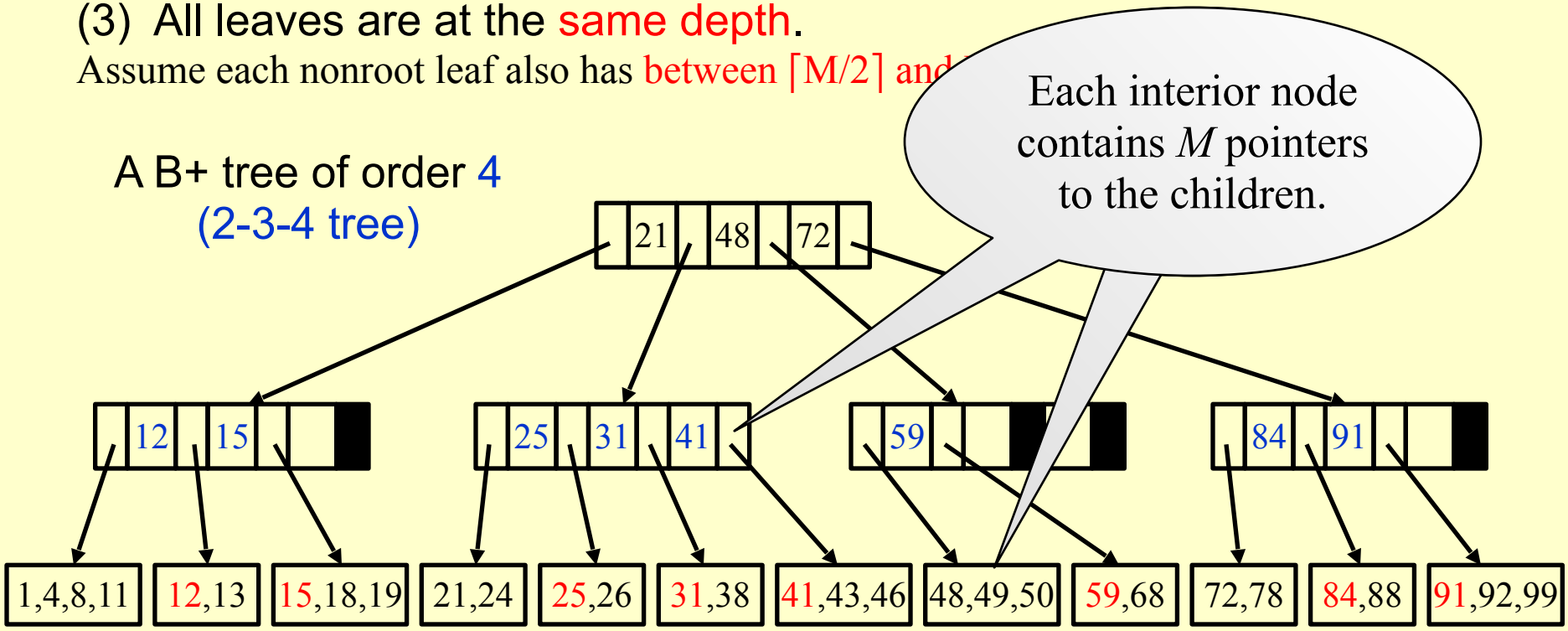
(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

Assume each nonroot leaf also has between ⌈M/2⌉ and

A B+ tree of order 4
(2-3-4 tree)



# B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

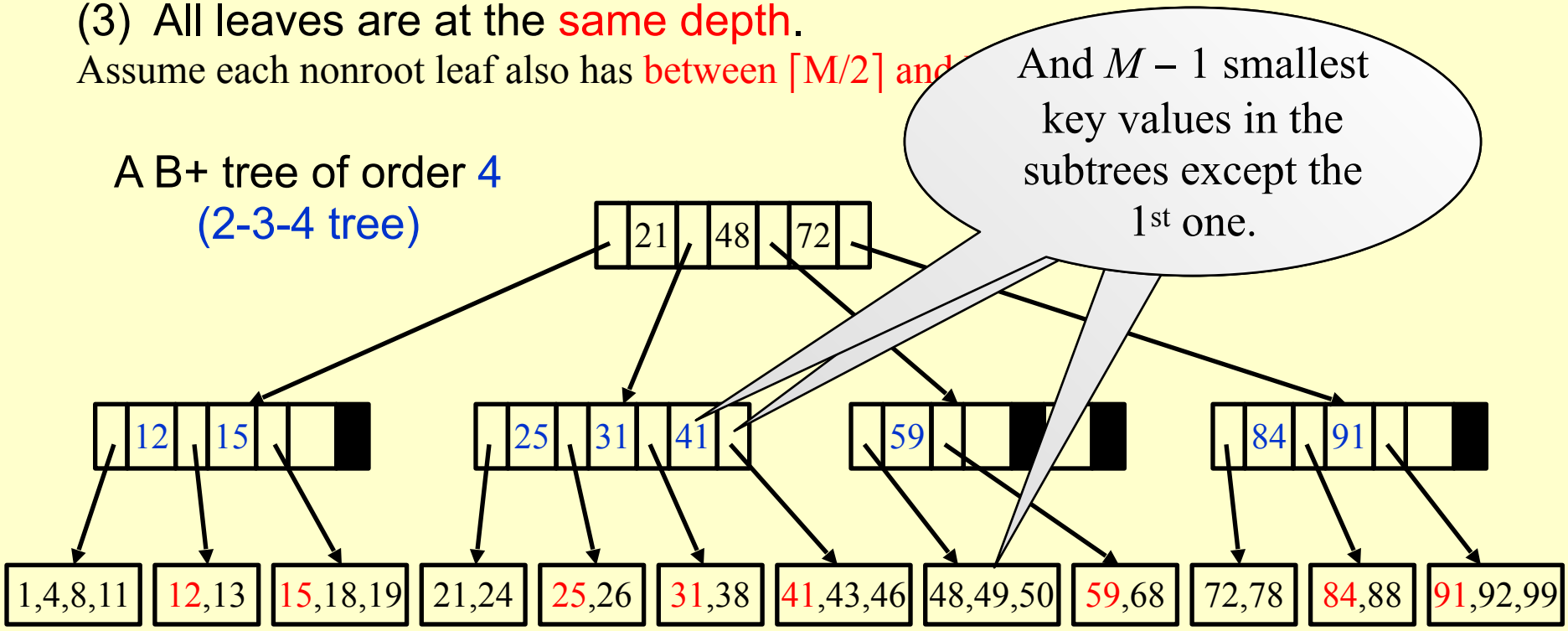
(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

Assume each nonroot leaf also has between ⌈M/2⌉ and

A B+ tree of order 4
(2-3-4 tree)



# B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

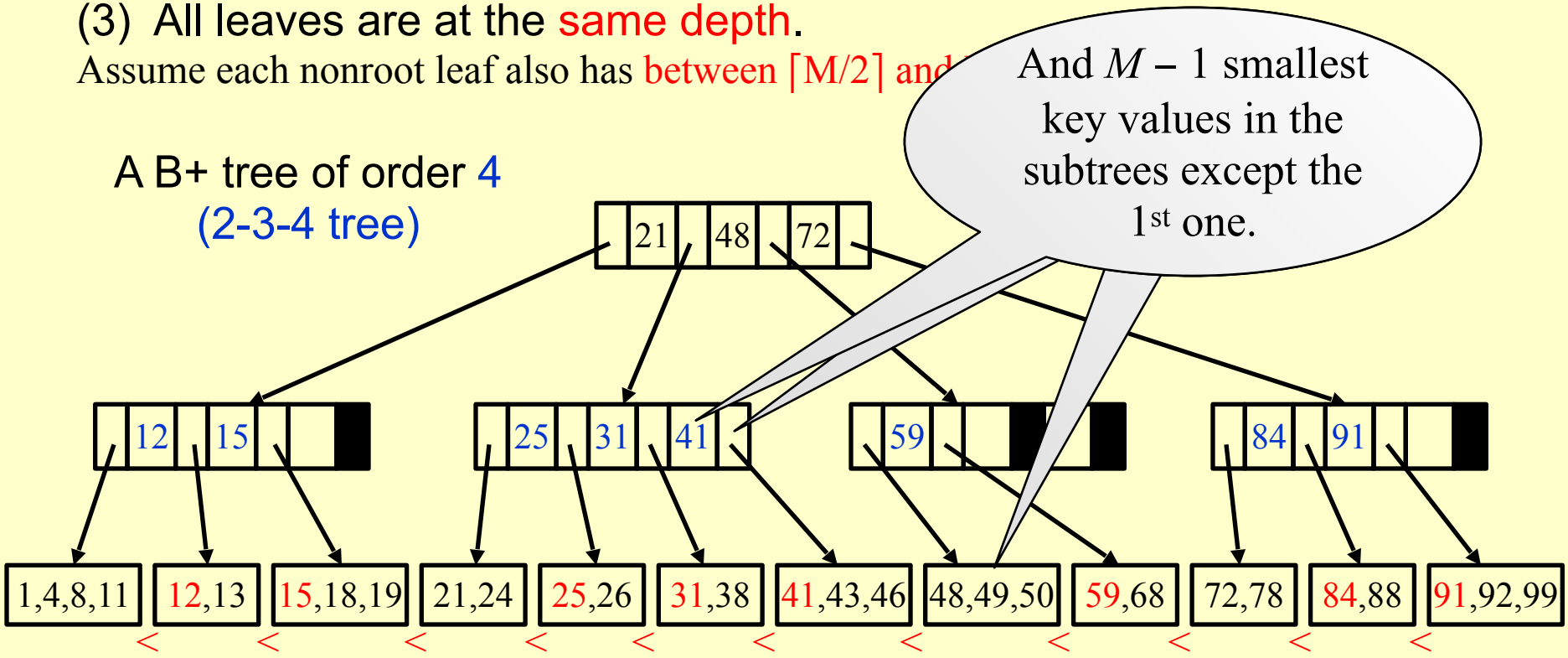
(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

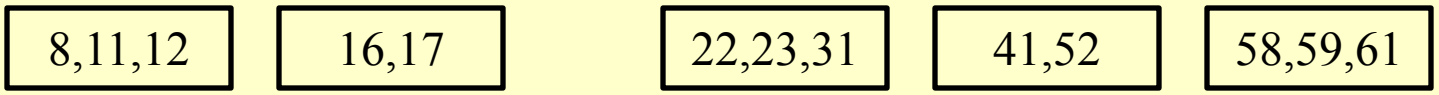
Assume each nonroot leaf also has between ⌈M/2⌉ and

A B+ tree of order 4
(2-3-4 tree)

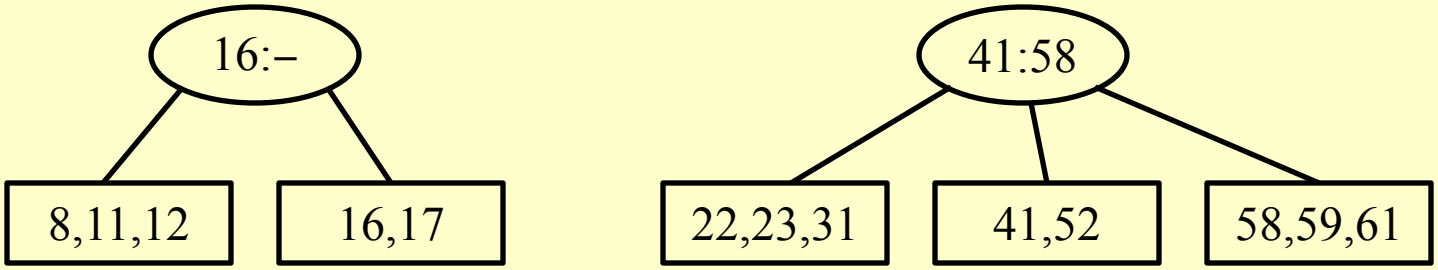


A B+ tree of order 3
(2-3 tree)

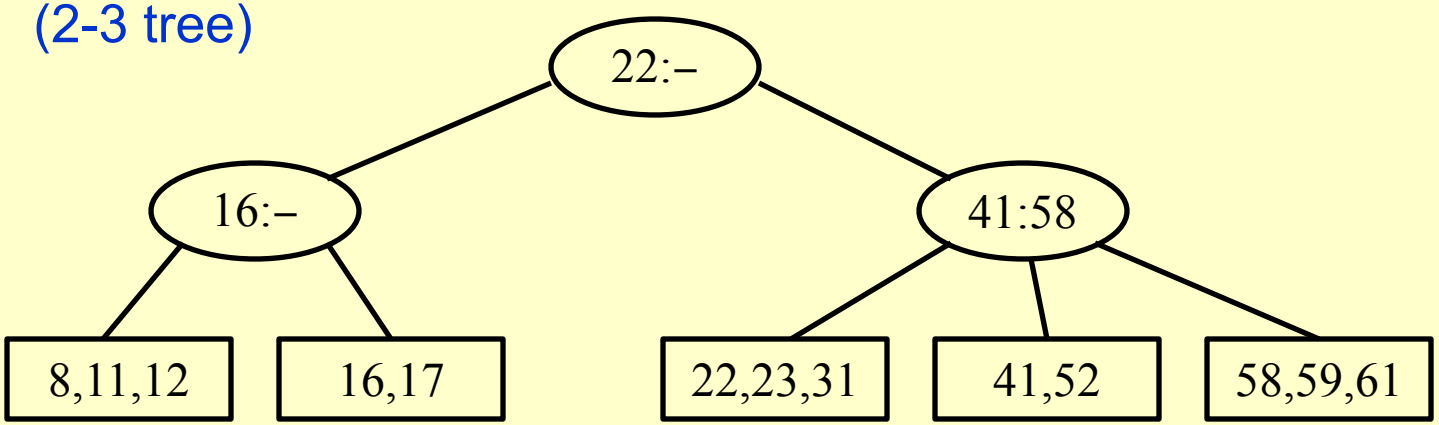
A B+ tree of order 3
(2-3 tree)



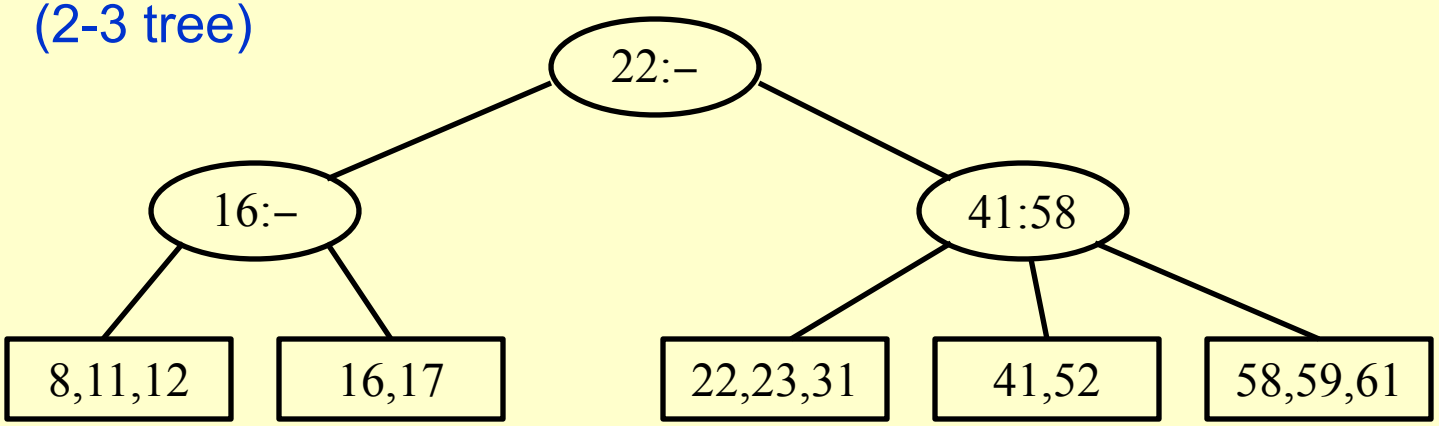
A B+ tree of order 3
(2-3 tree)



A B+ tree of order 3
(2-3 tree)

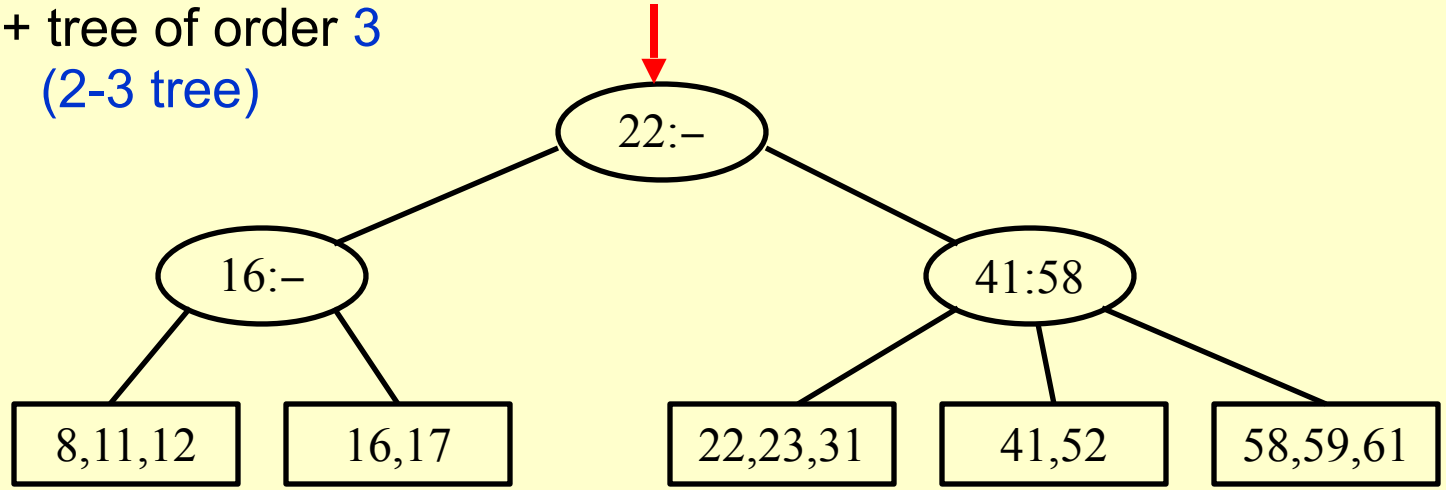


A B+ tree of order 3
(2-3 tree)



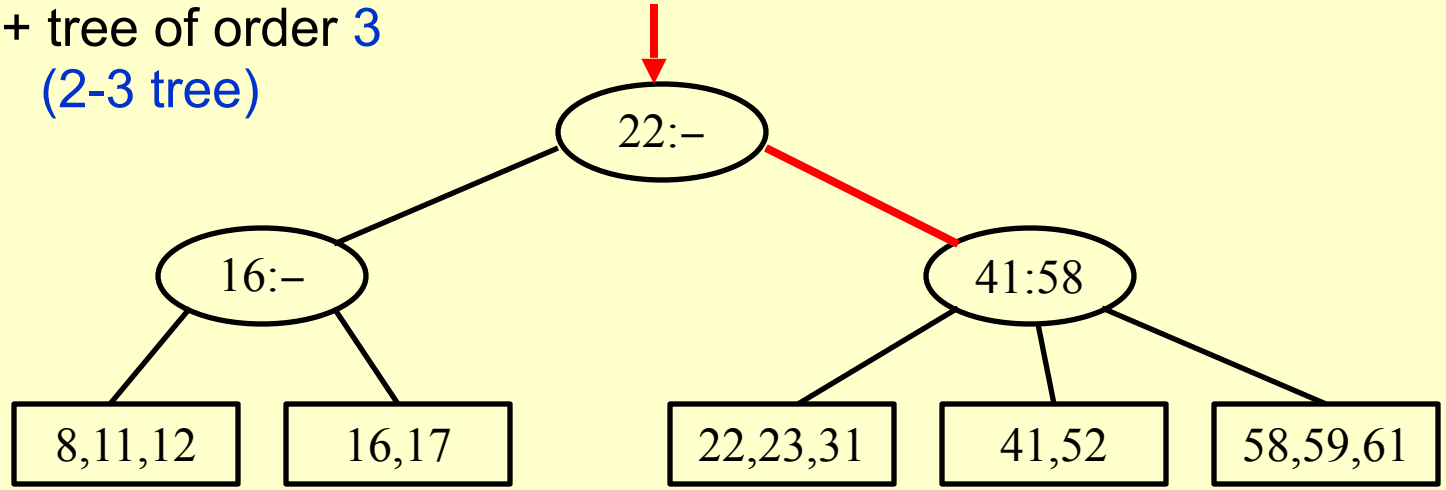

☞ Find: 52

A B+ tree of order 3
(2-3 tree)



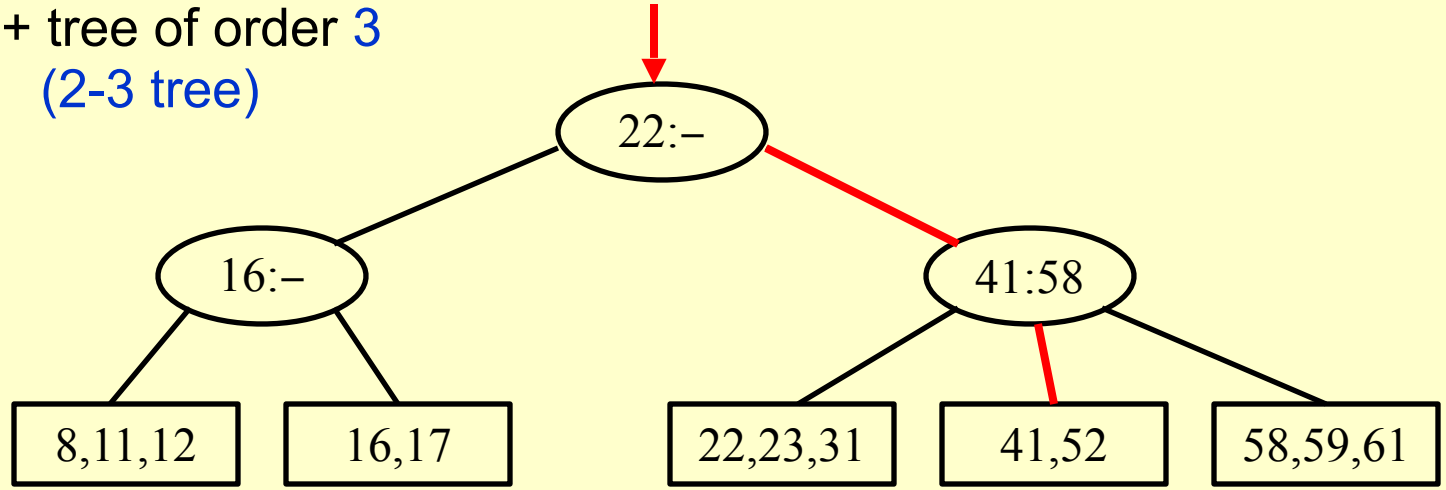

☞ Find: 52

A B+ tree of order 3
(2-3 tree)



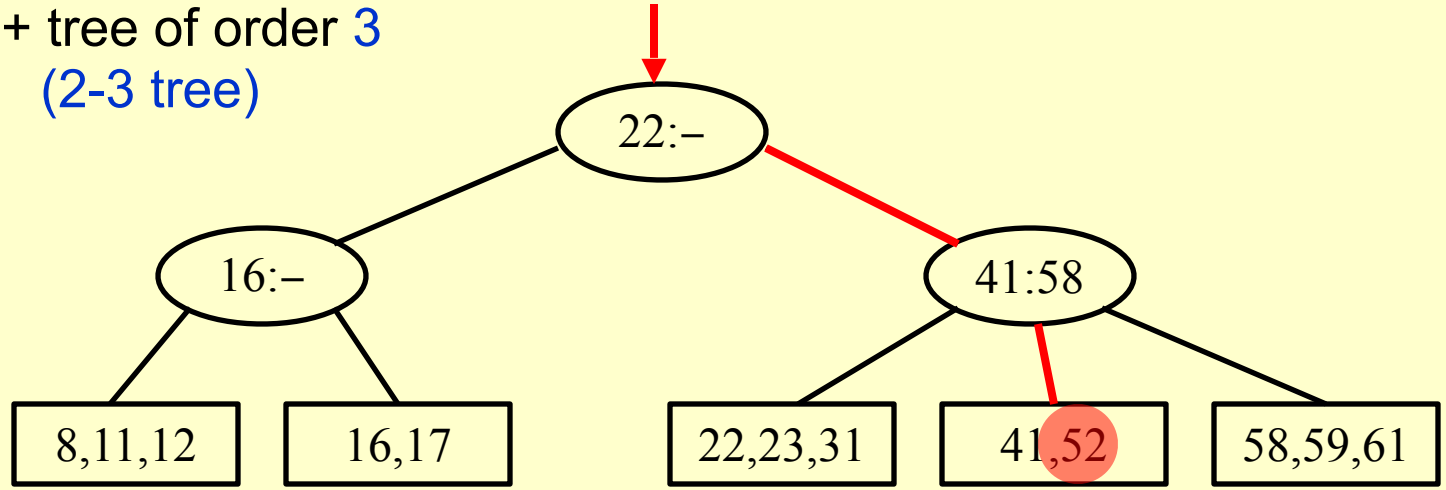

☞ Find: 52

A B+ tree of order 3
(2-3 tree)



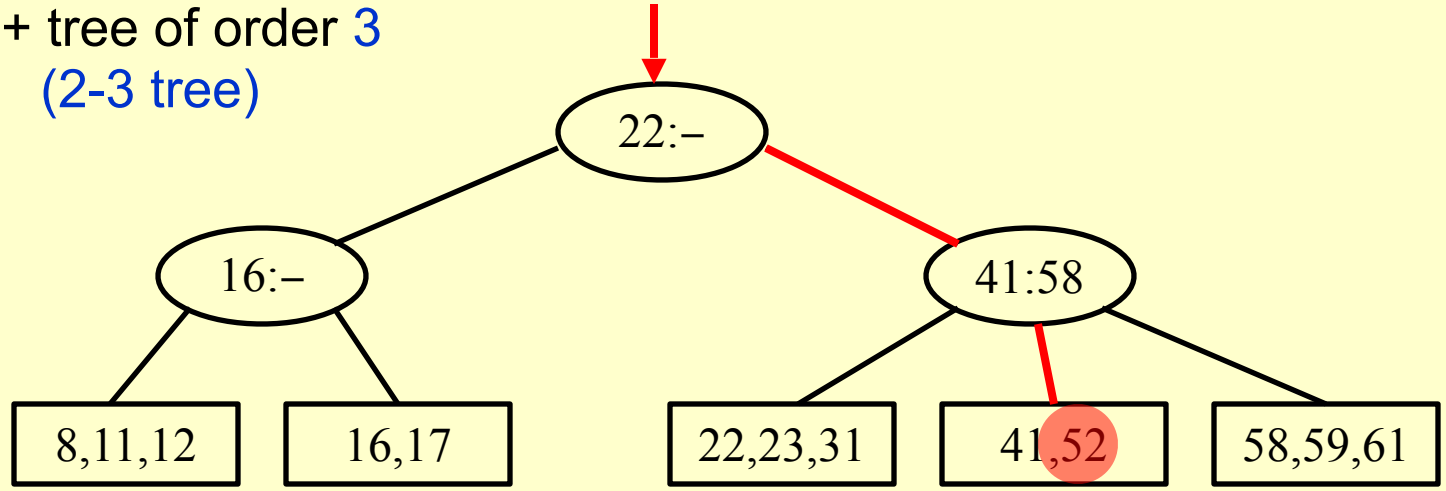

☞ Find: 52

A B+ tree of order 3
(2-3 tree)



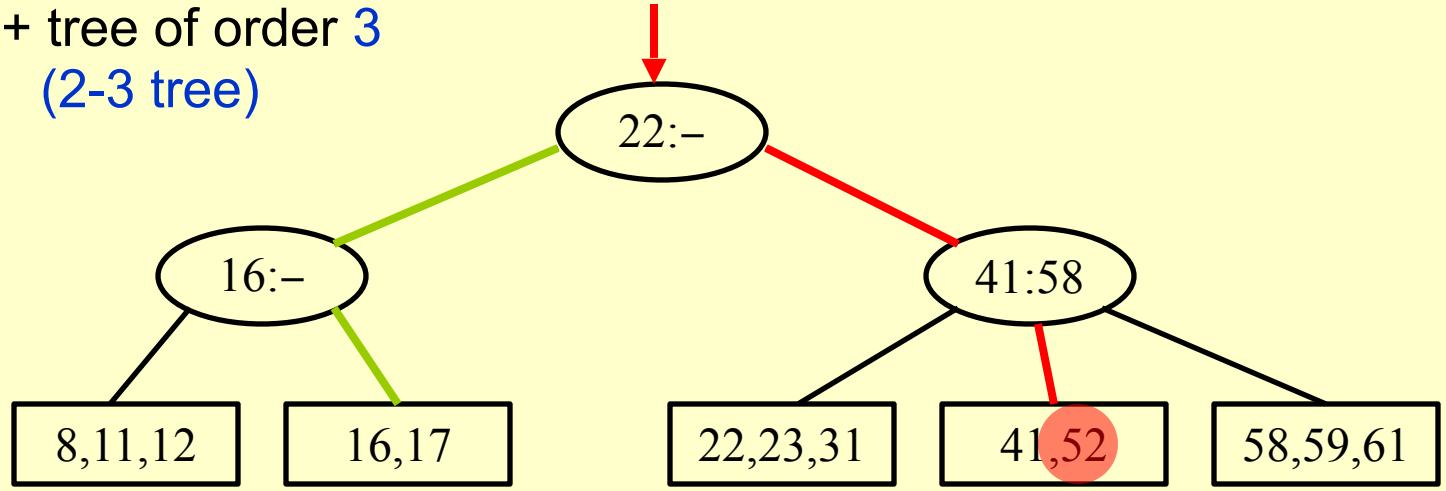

☞ Find: 52

A B+ tree of order 3
(2-3 tree)



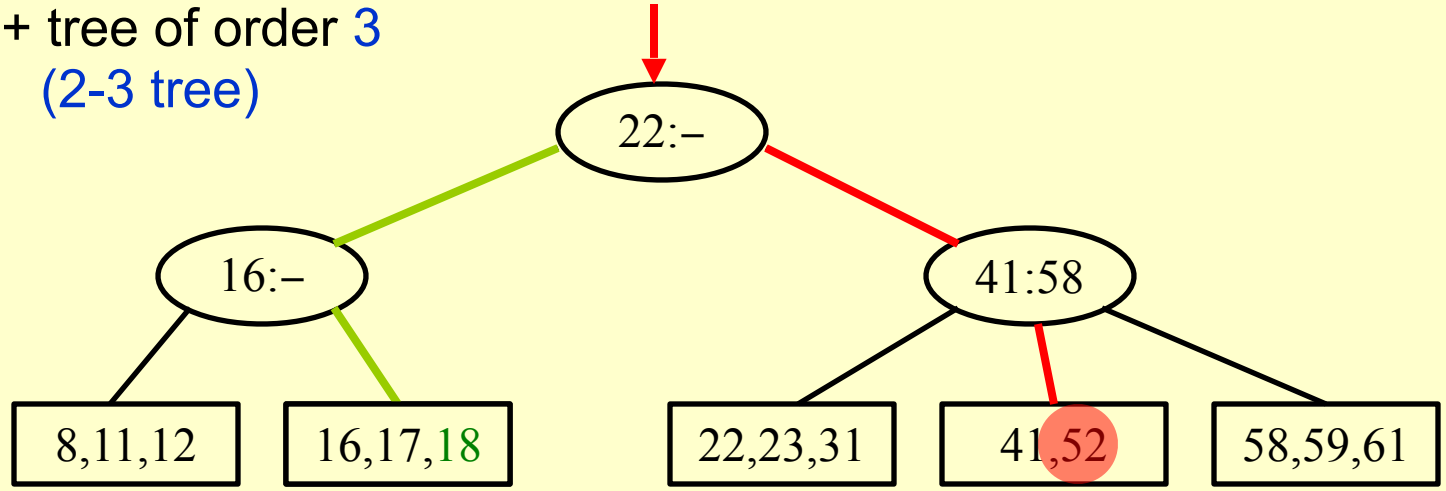

☞ Find: 52 ☞ Insert: 18

A B+ tree of order 3
(2-3 tree)



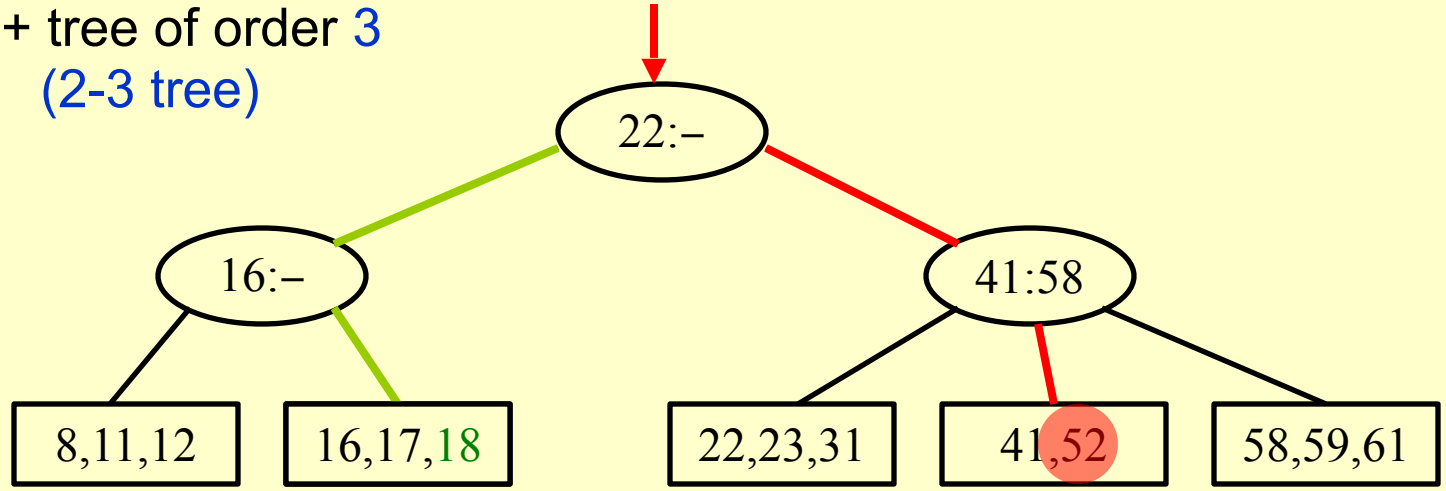

☞ Find: 52 ☞ Insert: 18

A B+ tree of order 3
(2-3 tree)



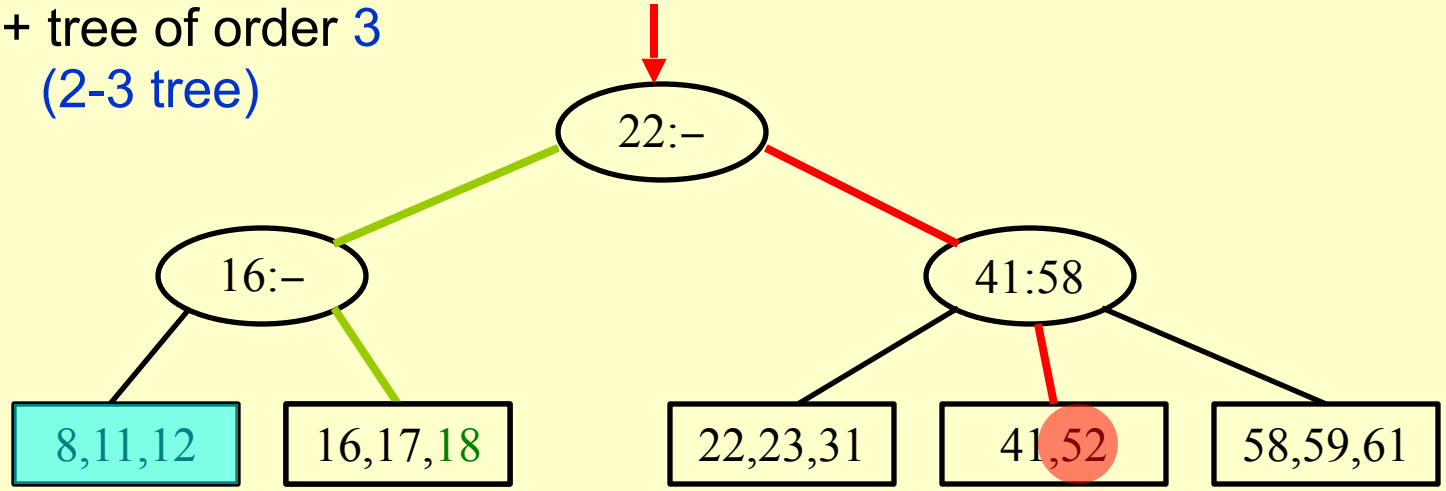

☞ Find: 52 ☞ Insert: 18

A B+ tree of order 3
(2-3 tree)



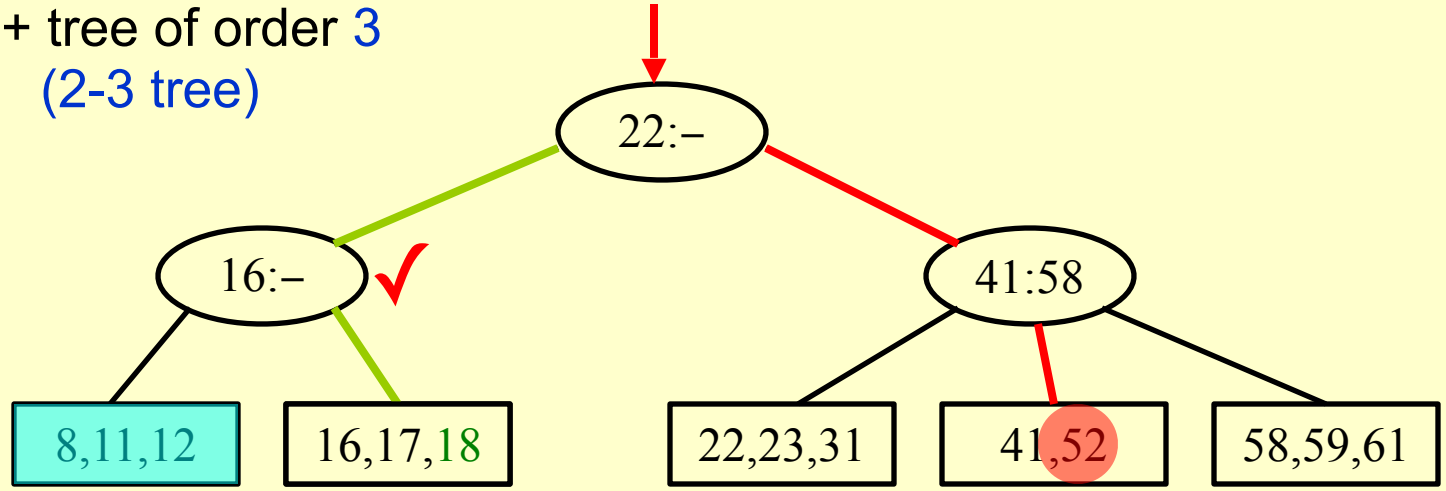

Find: 52 Insert: 18 Insert: 1

A B+ tree of order 3
(2-3 tree)



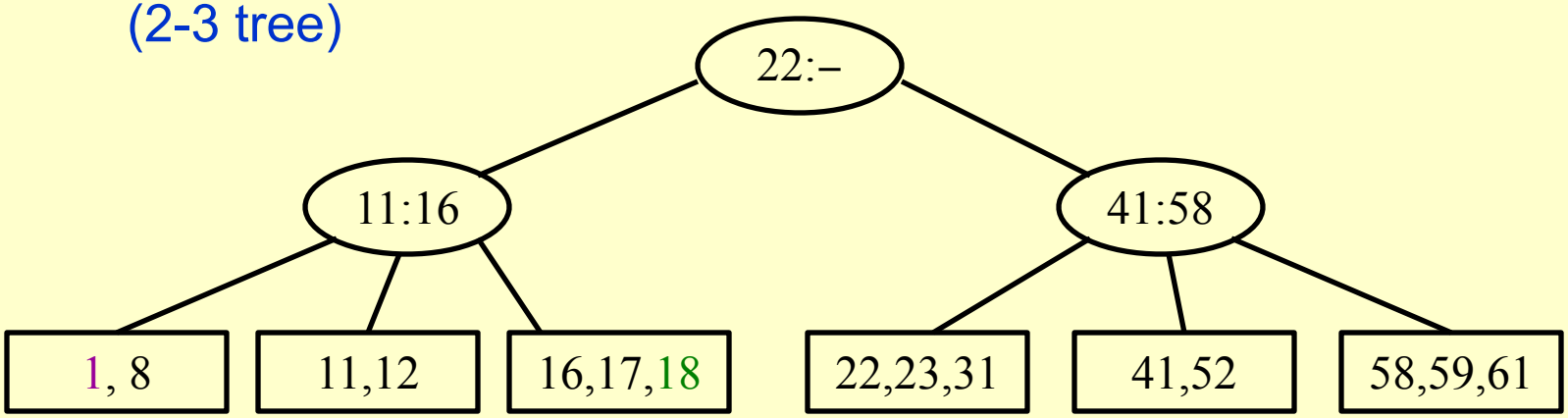

Find: 52 Insert: 18 Insert: 1

A B+ tree of order 3
(2-3 tree)



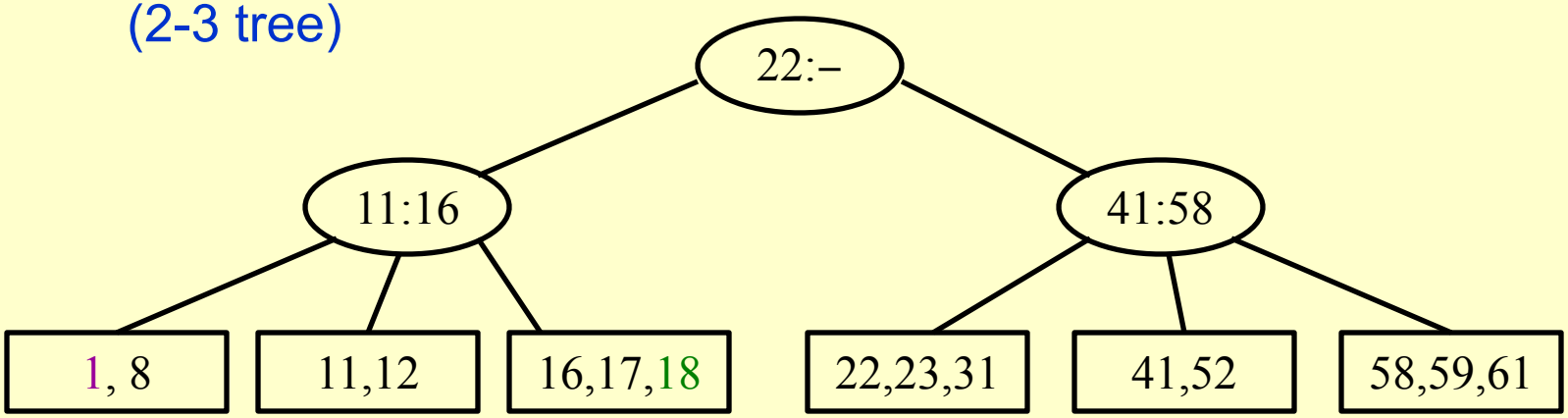

Find: 52 Insert: 18 Insert: 1

A B+ tree of order 3
(2-3 tree)



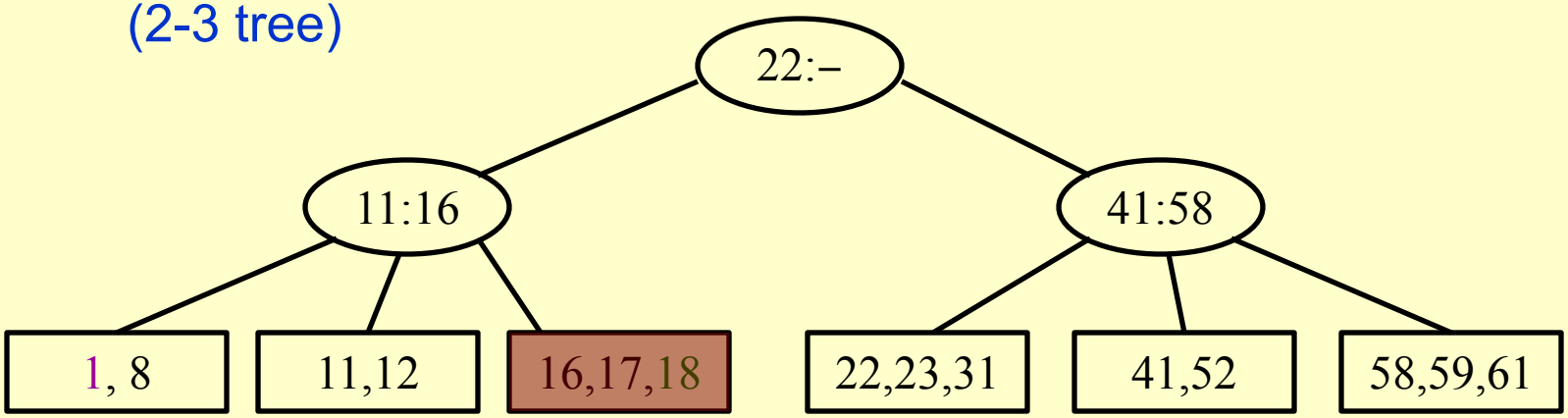

Find: 52 Insert: 18 Insert: 1

A B+ tree of order 3
(2-3 tree)



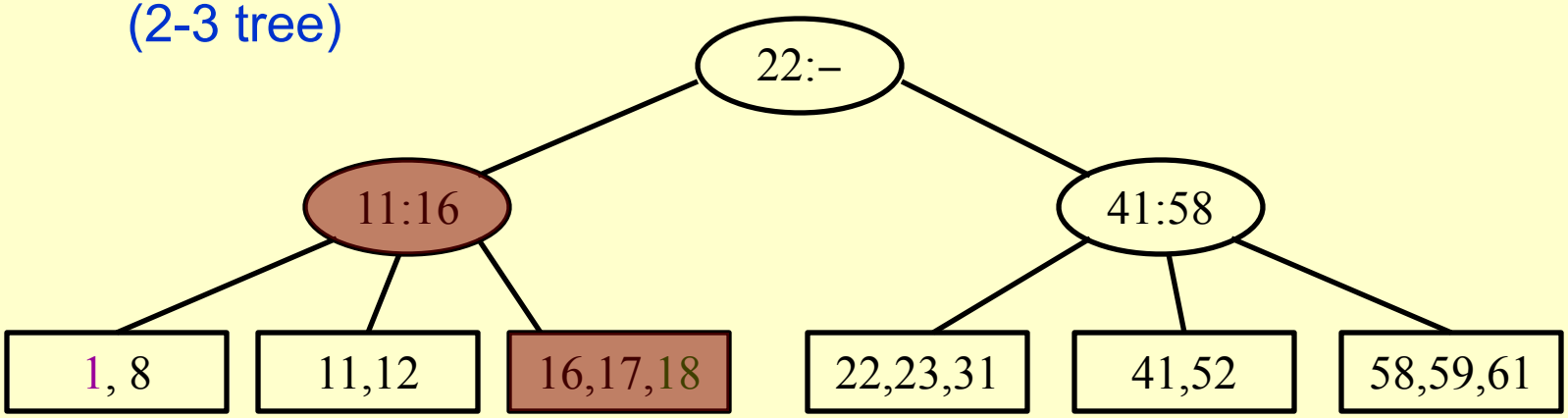

Find: 52 Insert: 18 Insert: 1 Insert: 19

A B+ tree of order 3
(2-3 tree)



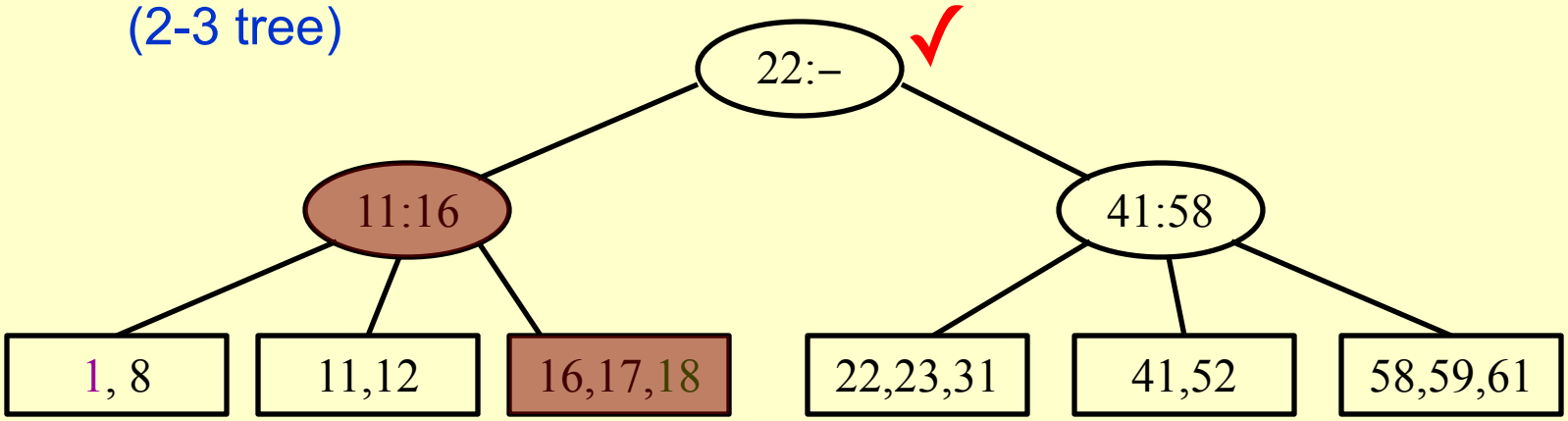

Find: 52 Insert: 18 Insert: 1 Insert: 19

A B+ tree of order 3
(2-3 tree)



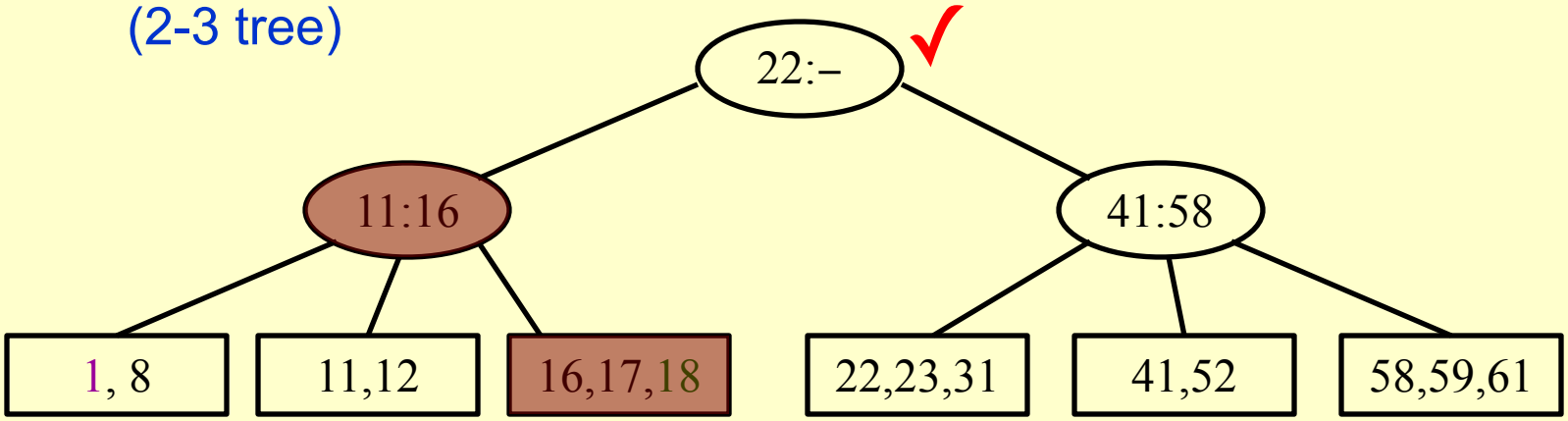

Find: 52 Insert: 18 Insert: 1 Insert: 19

A B+ tree of order 3
(2-3 tree)

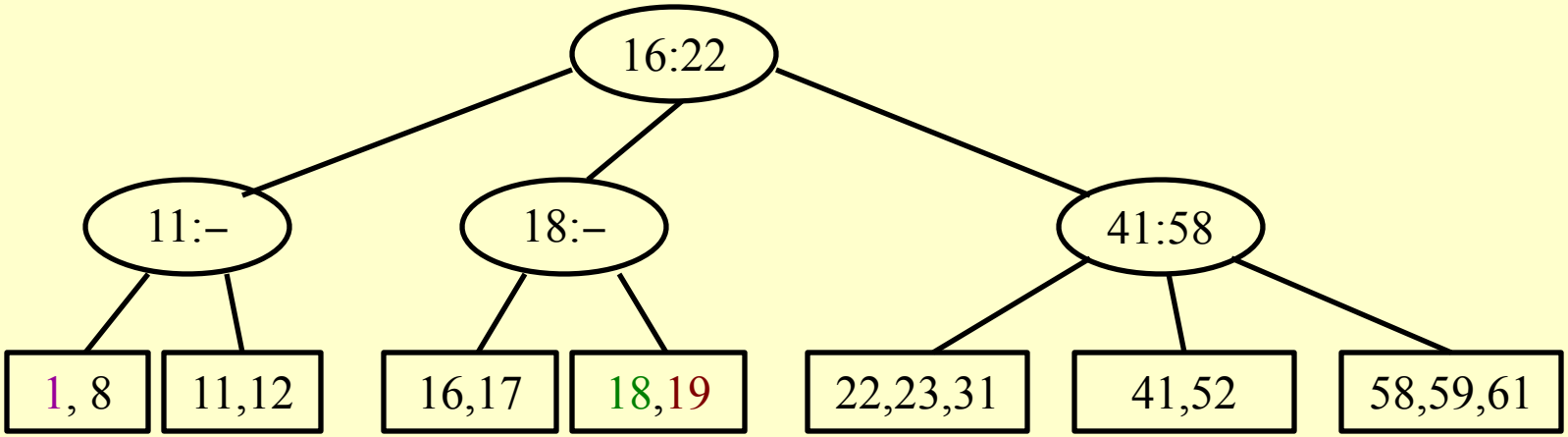



Find: 52 Insert: 18 Insert: 1 Insert: 19

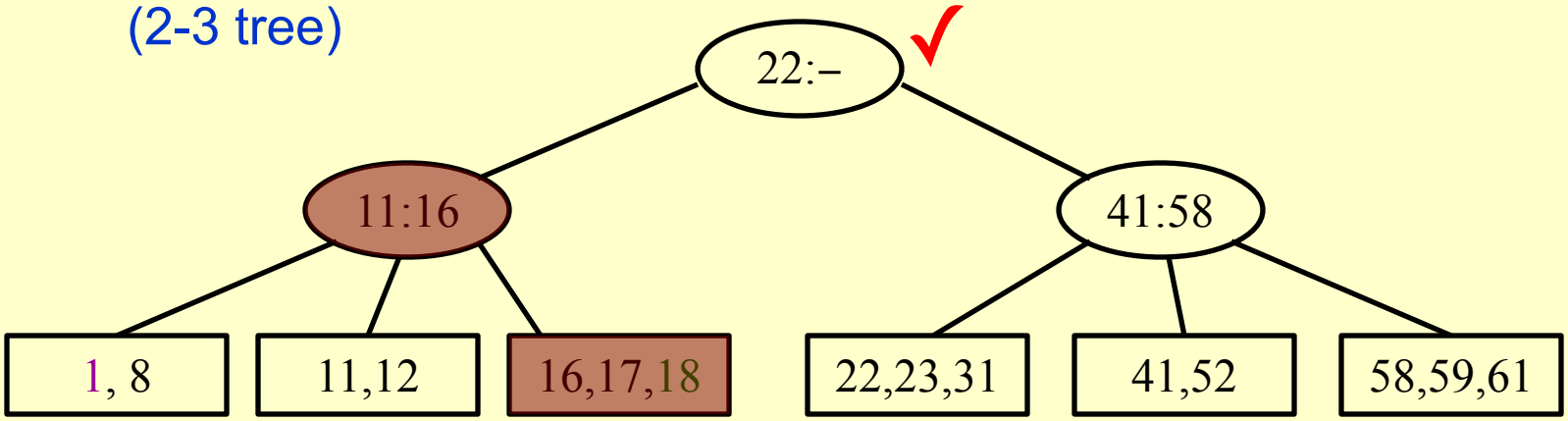
A B+ tree of order 3
(2-3 tree)



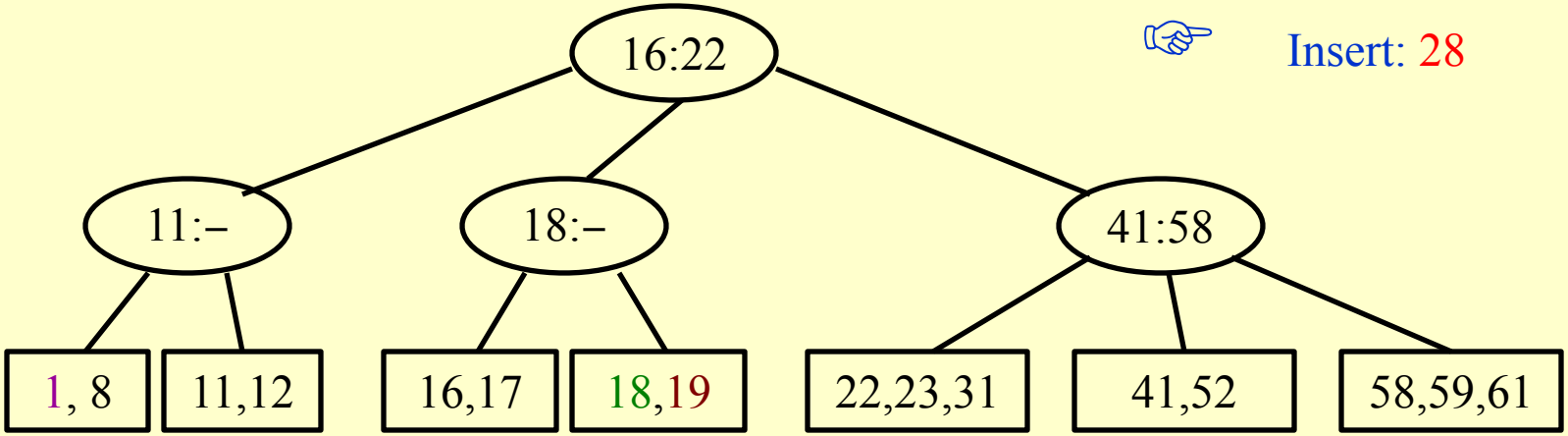

Find: 52 Insert: 18 Insert: 1 Insert: 19



A B+ tree of order 3
(2-3 tree)

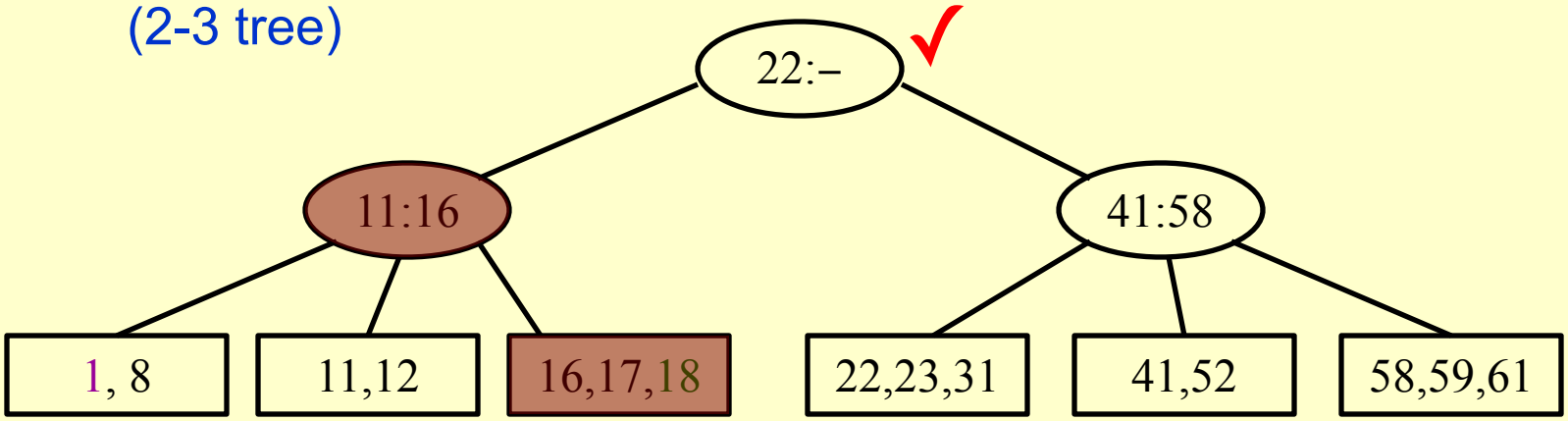



Find: 52 Insert: 18 Insert: 1 Insert: 19

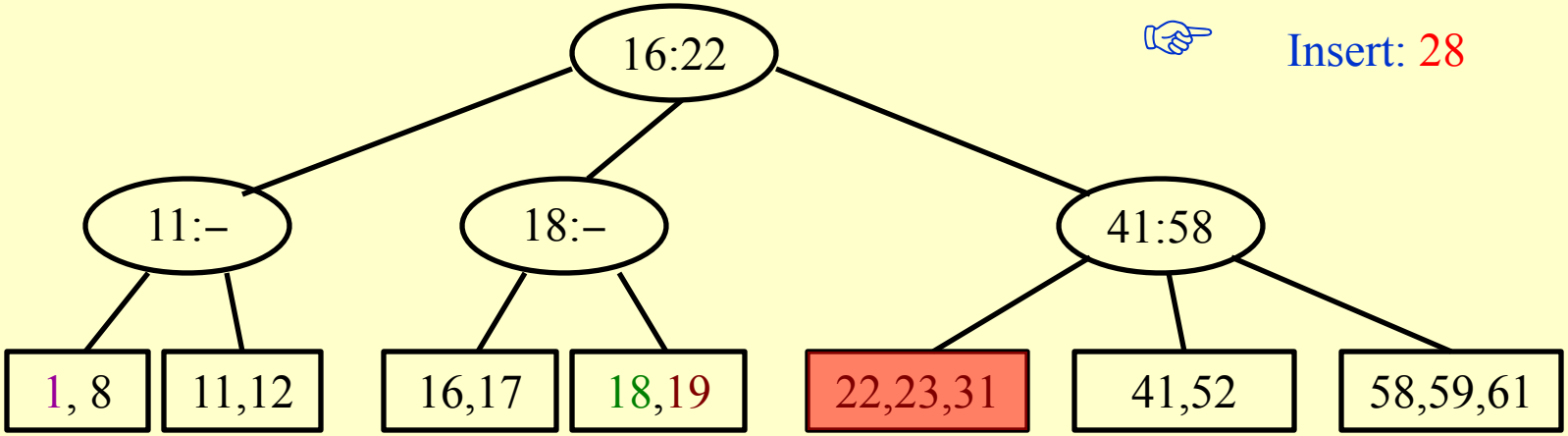



Insert: 28

A B+ tree of order 3
(2-3 tree)

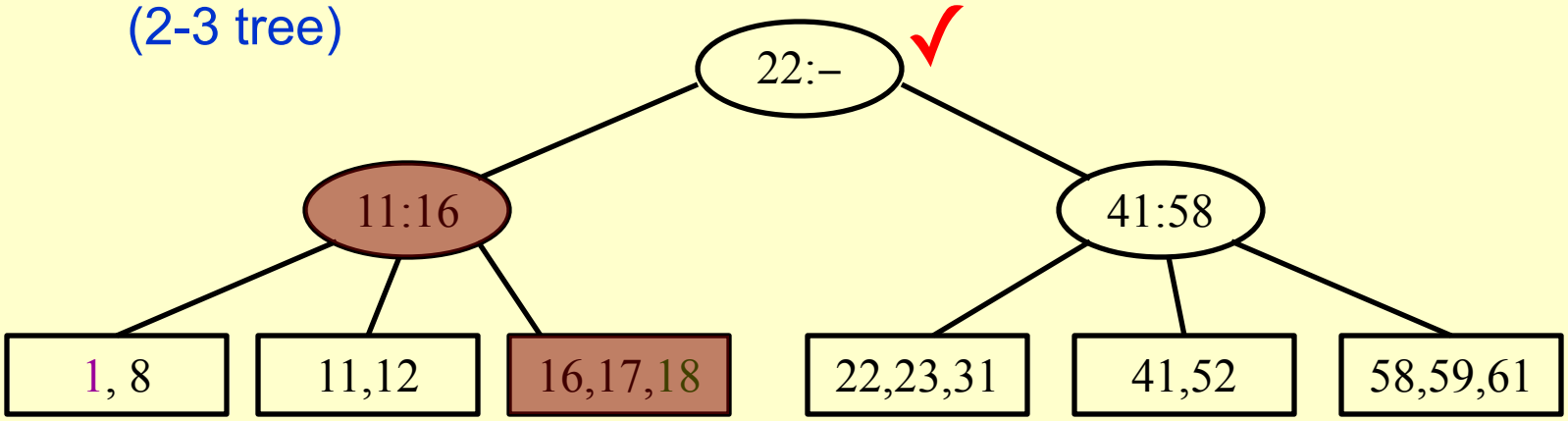



Find: 52 Insert: 18 Insert: 1 Insert: 19

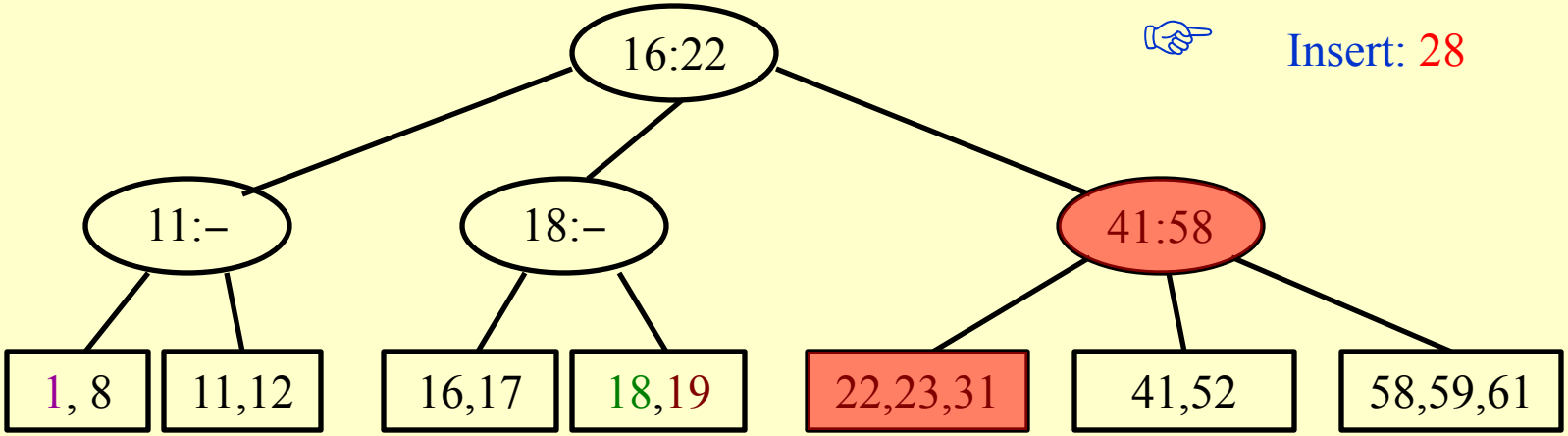



Insert: 28

A B+ tree of order 3
(2-3 tree)

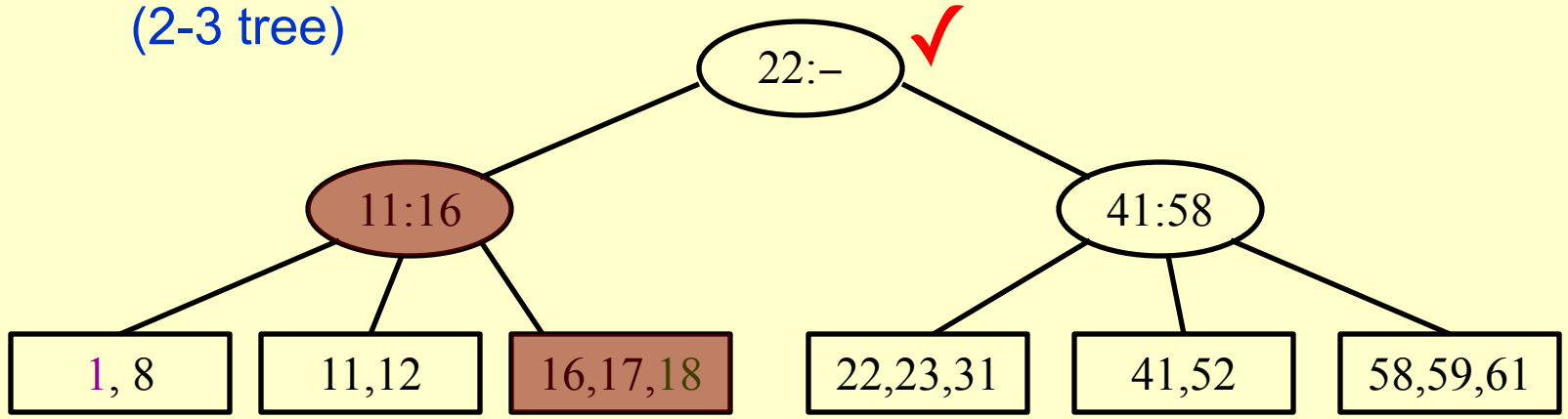



Find: 52 Insert: 18 Insert: 1 Insert: 19

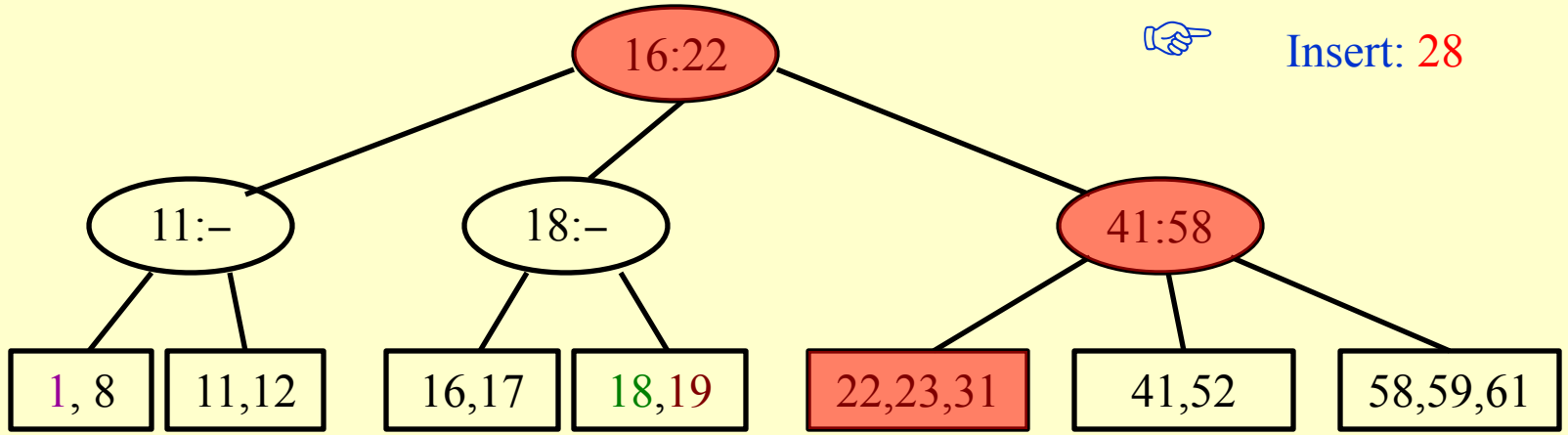



Insert: 28

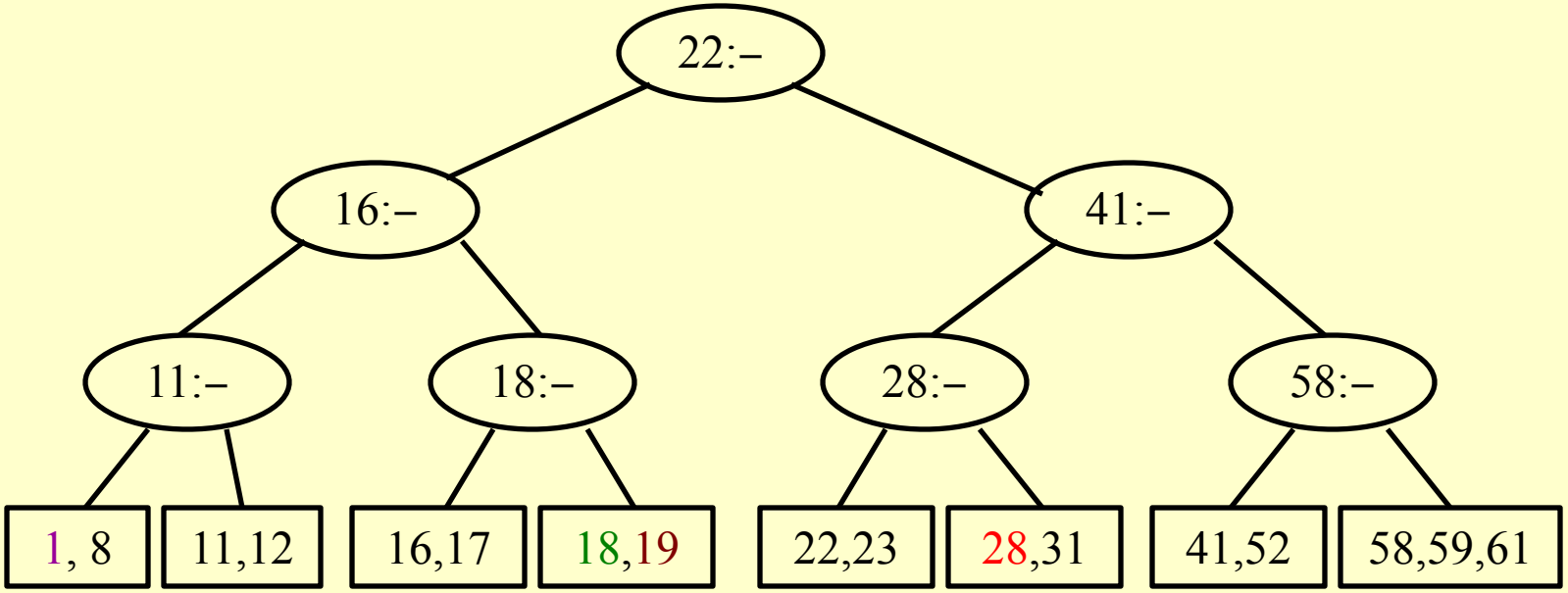
A B+ tree of order 3
(2-3 tree)

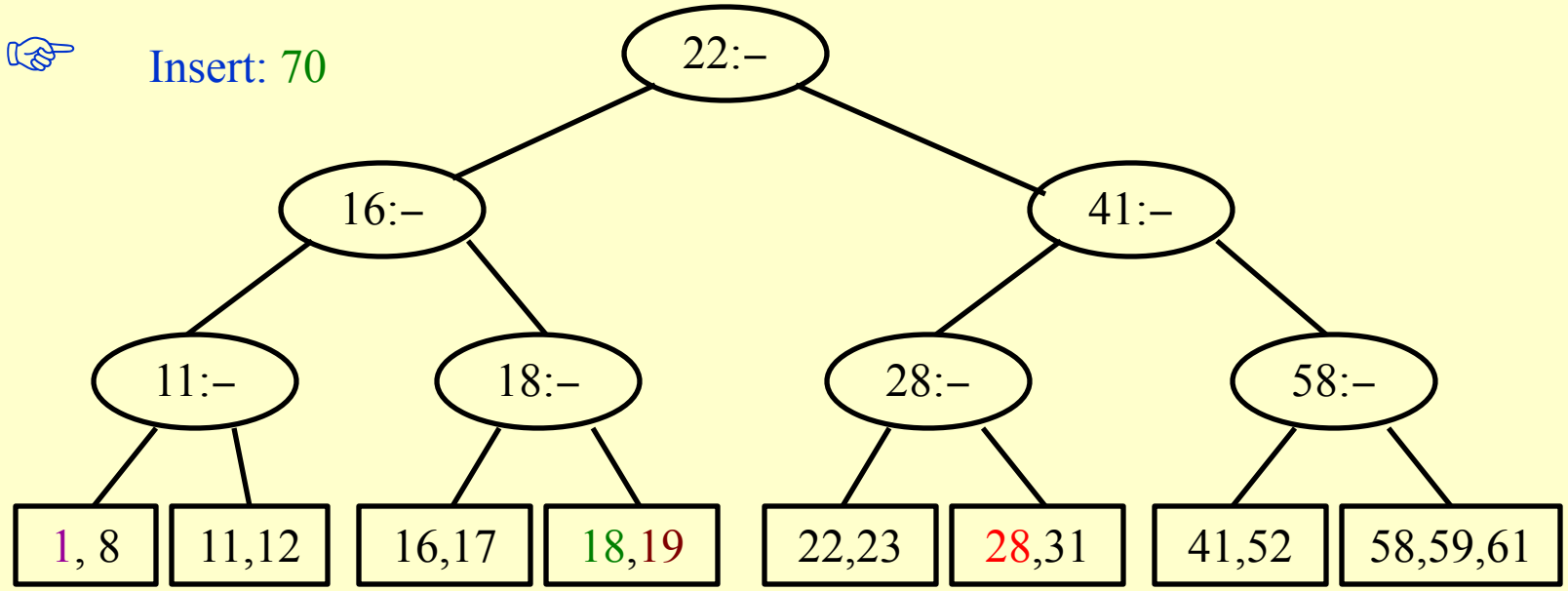


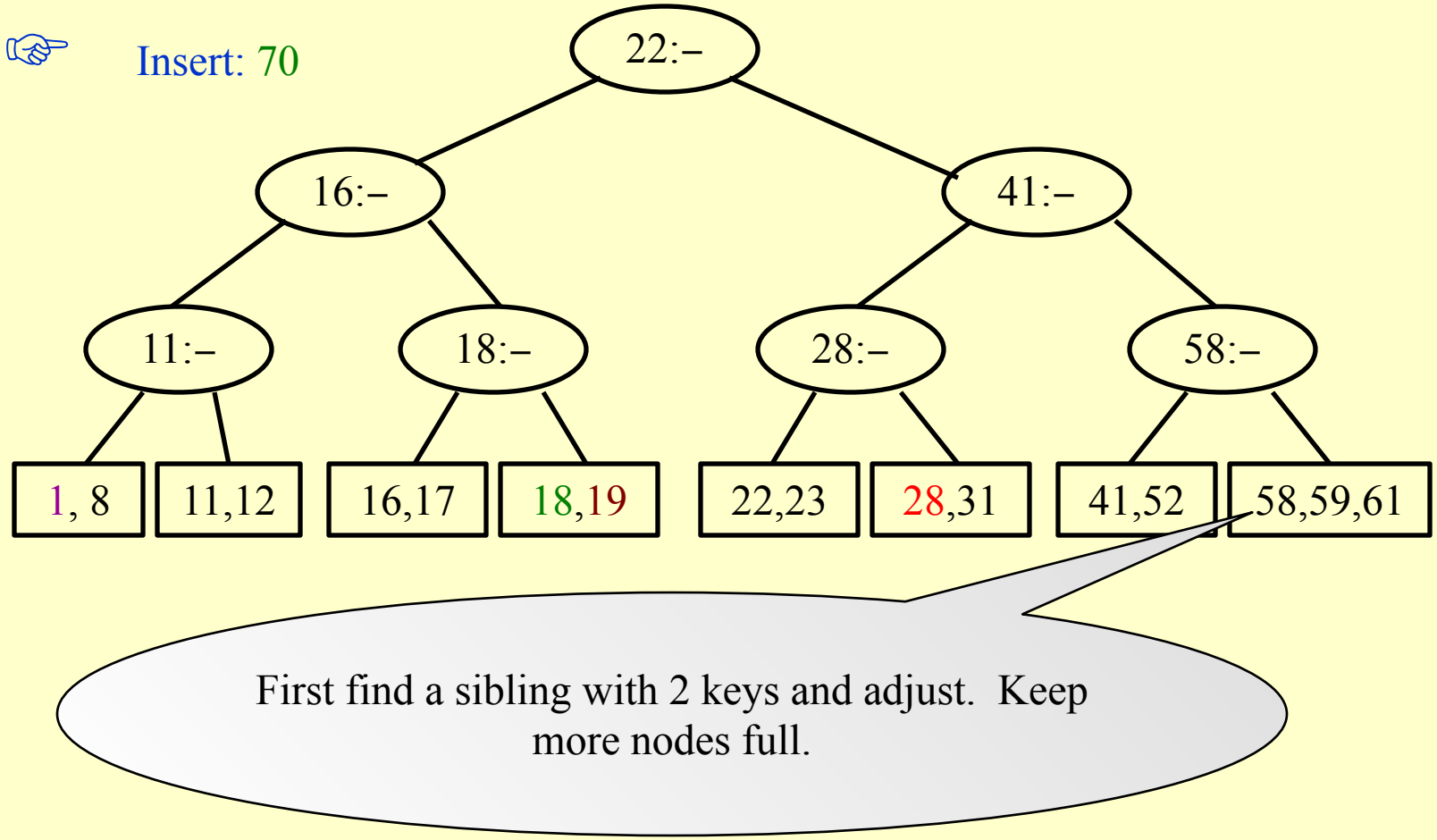

Find: 52 Insert: 18 Insert: 1 Insert: 19

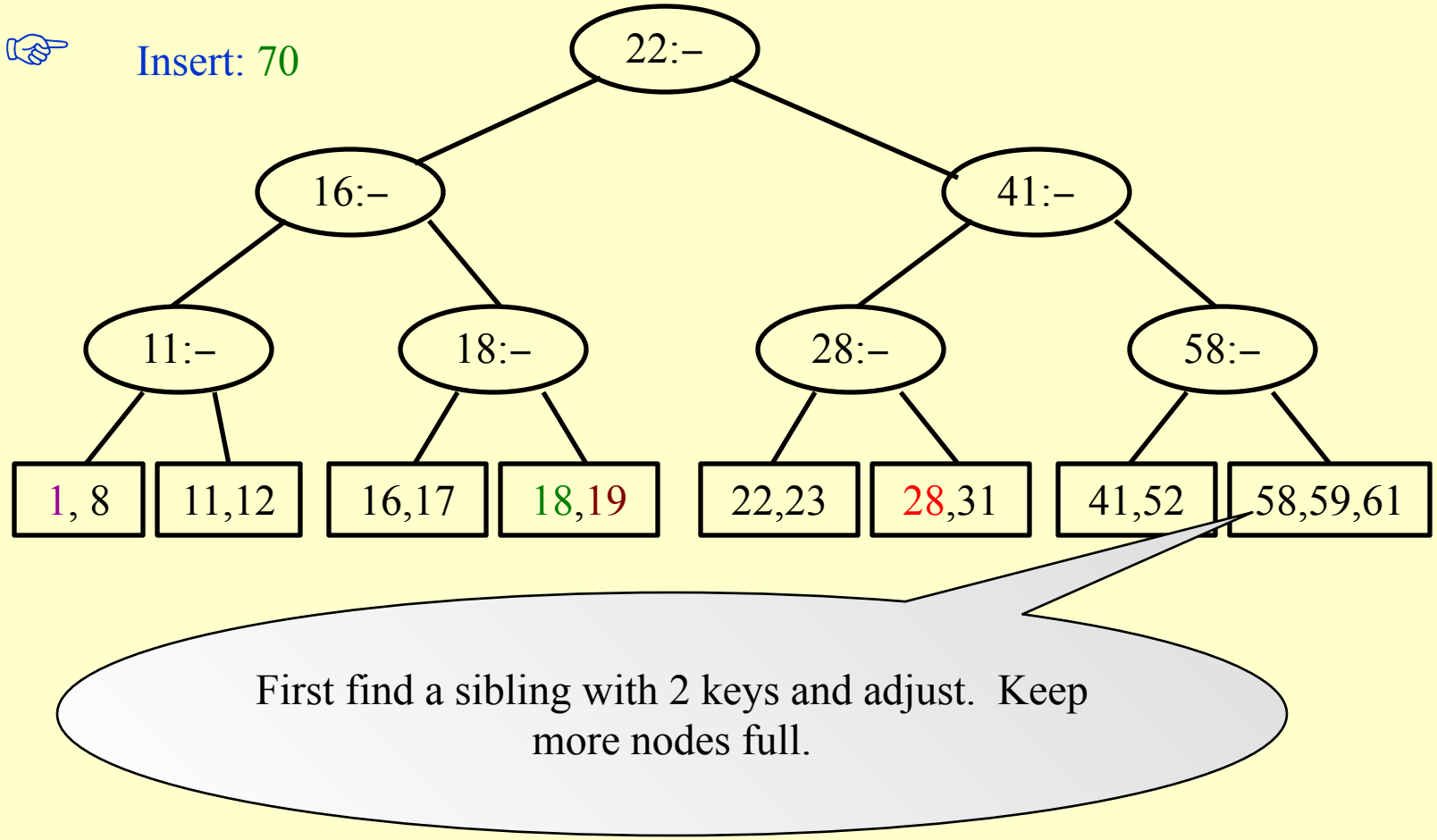



Insert: 28


22:–
16:–
41:–
11:–
18:–
28:–
58:–
1, 8
11,12
16,17
18,19
22,23
28,31
41,52
58,59,61


Insert: 70
22:–
16:–
41:–
11:–
18:–
28:–
58:–
1, 8
11,12
16,17
18,19
22,23
28,31
41,52
58,59,61


Insert: 70
22:–
16:–
41:–
11:–
18:–
28:–
58:–
1, 8
11,12
16,17
18,19
22,23
28,31
41,52
58,59,61
First find a sibling with 2 keys and adjust. Keep more nodes full.




Deletion is similar to insertion except that the root is removed when it loses two children.

For a general B+ tree of order M

For a general B+ tree of order M

```
Btree  Insert ( ElementType X,  Btree T )
{
    Search from root to leaf for X and find the proper leaf node;
    Insert X;
    while ( this node has M+1 keys ) {
            split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
    respectively;
            if (this node is the root)
                    create a new root with two children;
            check its parent;
    }
}
```

Depth($M$, $N$) =

For a general B+ tree of order M

```
Btree  Insert ( ElementType X,  Btree T )
{
    Search from root to leaf for X and find the proper leaf node;
    Insert X;
    while ( this node has M+1 keys ) {
            split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
    respectively;
            if (this node is the root)
                    create a new root with two children;
            check its parent;
    }
}
```

Depth($M$, $N$) = $O(\lceil \log_{\lceil M/2 \rceil} N \rceil)$

For a general B+ tree of order M

$T = O(M)$

```
Btree  Insert ( ElementType X,  Btree T )
{
    Search from root to leaf for X and find the proper leaf node;
    Insert X;
    while ( this node has M+1 keys ) {
        split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
    respectively;
        if (this node is the root)
            create a new root with two children;
        check its parent;
    }
}
```

Depth($M$, $N$) = $O(\lceil \log_{\lceil M/2 \rceil} N \rceil)$

For a general B+ tree of order M

$T = O(M)$

```
Btree  Insert ( ElementType X,  Btree T )
{
    Search from root to leaf for X and find the proper leaf node;
    Insert X;
    while ( this node has M+1 keys ) {
            split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
    respectively;
            if (this node is the root)
                    create a new root with two children;
            check its parent;
    }
}
```

$T(M, N) = O( (M/\log M) \log N )$

Depth($M$, $N$) = $O( \lceil \log_{\lceil M/2 \rceil} N \rceil )$

For a general B+ tree of order M

$T = O(M)$

```
Btree  Insert ( ElementType X,  Btree T )
{
    Search from root to leaf for X and find the proper leaf node;
    Insert X;
    while ( this node has M+1 keys ) {
            split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
    respectively;
            if (this node is the root)
                    create a new root with two children;
            check its parent;
    }
}
```

$T(M, N) = O( (M/\log M) \log N )$

Depth($M$, $N$) = $O( \lceil \log_{\lceil M/2 \rceil} N \rceil )$

$T_{Find}(M, N) = O( \log N )$

For a general B+ tree of order M

$T = O(M)$

```
Btree  Insert ( ElementType X,  Btree T )
{
    Search from root to leaf for X and find the proper leaf node;
    Insert X;
    while ( this node has M+1 keys ) {
            split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
    respectively;
            if (this node is the root)
                    create a new root with two children;
            check its parent;
    }
}
```

$T(M, N) = O( (M/\log M) \log N )$

Depth($M$, $N$) = $O( \lceil \log_{\lceil M/2 \rceil} N \rceil )$

$T_{Find}(M, N) = O( \log N )$

Note: The best choice of M is 3 or 4.

# Historial Notes

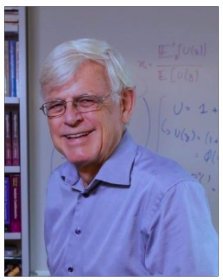
- 2-3-4 tree (1972) and B-tree (1970):

Rudolf Bayer

Edward M. McCreight

- Red-black tree (1978):

Leonidas J. Guibas



Robert Sedgewick

- 2-3 tree (1970):

John Hopcroft

# Balanced Search Trees (II)

- Review of amortized analysis
- Red-black trees
- B+ trees
- Take-home messages

# Take-Home Messages

- Red-black trees:
  - Binary search tree version of 2-3-4 trees. The red nodes are for represent >2 branches in each node.
  - The major properties lie in that the black height is balanced for each node.
  - The insertion and deletion involve constant cost on rotations.
- B-trees:
  - Search trees with more branches. Suitable for reducing access cost on nodes, applications on database, secondary drives…
  - Reduce tree depth by increasing the number of branches.

# Balanced Search Trees

- AVL trees: suitable when look-up costs matter most.
- Splay trees: suitable when the same items are visited repeatedly.
- Red-black trees: suitable when insertion/deletion costs matter most.
- B-trees: suitable when the data are stored in blocks, and the access costs matter most.

Thanks for your attention!
Discussions?

# Reference

Introduction to Algorithms (4th Edition): Chap.13, 18.

Algorithms (4th Edition): Chap. 3.3.

http://www.cs.columbia.edu/~bauer/cs3134-f15/slides/w3134-1-lecture11.pdf